# Dynamic Scheduling in High-Level Compilation for Adaptive Computers

Von der Carl-Friedrich-Gauß-Fakultät
Technische Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von Dipl.-Inform. Hagen Gädke-Lütjens

geboren am 15.02.1979

in Berlin

| | |
|---|---|
| Eingereicht am: | 11.03.2011 |
| Mündliche Prüfung am: | 27.04.2011 |
| Referent: | Prof. Dr. Ulrich Golze |
| Korreferent: | Prof. Dr. Andreas Koch |

(2011)

# Abstract

The single-thread performance of conventional CPUs has not improved significantly due to the stagnation of the CPU frequencies since 2003. Adaptive computers, which combine a CPU with a reconfigurable hardware unit (e.g., an FPGA) used as hardware accelerator, represent a promising, alternative compute platform. During the past 10 years, much research has been done to develop tools that enhance the usability of adaptive computers. An important goal here is the development of an adaptive compiler, which compiles hardware descriptions from common high-level languages such as C in a fully automated way.

Most of the compilers developed until today use static scheduling for the generated hardware. However, for complex programs containing nested loops, irregular control flow, and arbitrary pointers, *dynamic* scheduling is more appropriate. Unexpected operator lantencies which occur, e.g., when accessing cached memories, force a statically scheduled system to stall *all* operations in the design. Using dynamic scheduling, only the directly affected operations stall, while independent operators can continue their computation. This work examines the feasibility of compiling to dynamically scheduled hardware, an approach that has been the subject of only limited research efforts so far.

Based on previous work we have developed the adaptive compiler COMRADE 2.0, which generates synthesizable hardware descriptions (using dynamic scheduling) from ANSI C. The compiler front-end transforms the input program into a structured control flow graph, which is partitioned into hardware and software components. For each hardware component, COMRADE 2.0 creates a data flow-based COMRADE Controller Micro-Architecture (COCOMA) instance. COCOMA models even complex control and memory dependences and is thus especially suitable as intermediate representation in a compile flow that supports complex C programs. COCOMA allows pipelining, speculative execution with early evaluation, and even cancelation of misspeculated operations with cancel tokens (CTs). In this context, COCOMA supports two different token flow models: static and dynamic CTs. From the COCOMA model, the compiler back-end generates synthesizable hardware descriptions (hardware kernels, each consisting of a data path and a sequencer)

in the hardware description language Verilog. The data path instantiates hardware operators provided by a platform-independent hardware operator library, which has also been developed in the context of this work.

We examine the effects of parameter variations and low-level optimizations on the simulation and synthesis results. Our analysis shows that static CTs outperform dynamic CTs, because they save area resources without affecting the runtime. Operation chaining can help to eliminate unnecessary registers and thus save runtime as well as area resources. Further runtime savings are possible by memory access reordering and parallelization. The most promising optimization technique considering the runtime is memory localization which can significantly increase the memory bandwidth available to the compiled hardware kernels. Using memory localization we have obtained hardware kernel speed-ups of up to 37x over an embedded CPU. Beyond single kernels, we have also examined the application-level speed-ups. For a multi-phase image compression application, we have managed a speed-up of 5x over a 400 MHz superscalar embedded CPU.

# Kurzfassung

Bedingt durch die Stagnation der CPU-Frequenzen stagniert seit 2003 auch die Single-Thread-Rechenleistung herkömmlicher CPUs. Adaptive Rechner bieten eine vielversprechende, alternative Rechenarchitektur, indem sie die CPU um eine rekonfigurierbare Einheit (z. B. einen FPGA), die als Hardware-Beschleuniger verwendet wird, erweitern. In den vergangenen zehn Jahren wurde viel Forschung in Entwurfswerkzeuge investiert, die eine einfachere und praktikablere Verwendung adaptiver Rechner ermöglichen sollen. Ein wesentliches Ziel ist dabei die Entwicklung eines adaptiven Compilers, der aus einer allgemein verwendeten Hochsprache wie C vollautomatisch Hardwarebeschreibungen erzeugen kann.

Die meisten der bisher entwickelten Compiler setzen in der erzeugten Hardware statisches Scheduling ein. Für komplexere Programme mit verschachtelten Schleifen, irregulärem Kontrollfluss und beliebigen Zeigerzugriffen ist jedoch *dynamisches* Scheduling besser geeignet. Unerwartete Operatorlatenzen, wie z. B. ein Cache-Miss bei Speicherzugriffen, erzwingen bei statischem Scheduling das Anhalten aller Operationen in der Hardware-Recheneinheit, während bei dynamischem Scheduling nur direkt betroffene Operationen angehalten werden müssen, unabhängige Berechnungen aber weiterlaufen können. Diese Arbeit untersucht die praktische Machbarkeit des dynamischen Schedulings, zu dem es bislang kaum Untersuchungen im Kontext adaptiver Rechner gibt.

Auf der Grundlage bestehender Vorarbeiten haben wir den Compiler COMRADE 2.0 entwickelt, der aus ANSI C vollautomatisch synthetisierbare Hardwarebeschreibungen generieren kann, die dynamisches Scheduling verwenden. Das Compiler-Front-End erzeugt aus dem Eingabeprogramm einen strukturierten Kontrollflussgraphen, der in Hardware- und Software-Komponenten partitioniert und optimiert wird. Für jede Hardware-Komponente wird dann eine datenflussbasierte COMRADE Controller Micro-Architecture- (COCOMA-) Darstellung erzeugt, die auch komplexere Kontroll- und Speicherabhängigkeiten korrekt modelliert und daher besonders für die Compilierung komplexerer C-Programme geeignet ist. COCOMA ermöglicht Pipelining, spekulative Ausführung mit frühzeitiger Evaluation und sogar das Abbrechen unnötiger speku-

lativer Operationen mit Cancel-Tokens (CTs). Hierbei unterstützt COCOMA zwei verschiedene Tokenflussmodelle: statische und dynamische CTs. Aus der COCOMA-Darstellung erzeugt das Back-End dann synthetisierbare Hardwarebeschreibungen (HW-Kernels, bestehend aus Datenpfad und Steuerwerk) in der Hardware-Beschreibungssprache Verilog. Der Datenpfad instanziert Hardware-Operatoren aus einer plattformunabhängigen Operator-Bibliothek, die in weiten Teilen auch im Rahmen der hier vorgestellten Forschungen entstanden ist.

Wir untersuchen die Auswirkungen von Parameteränderungen und Low-Level-Optimierungen auf die Simulations- und Syntheseergebnisse. Unsere Untersuchungen zeigen, dass statische CTs dynamischen CTs überlegen sind, weil sie bei gleicher Laufzeit Ressourcen einsparen. Mit Operation-Chaining kann durch geschickte Entfernung von Pufferregistern gleichzeitig die Laufzeit verringert und die Fläche verkleinert werden. Weitere Laufzeitersparnisse ergeben sich durch die Umsortierung und Parallelisierung von Speicherzugriffen. Die für die Laufzeit vielversprechendste Optimierungstechnik ist die Speicherlokalisierung, die eine wesentlich höhere Speicherbandbreite bei geringer Latenz bietet. Wir messen hier Beschleunigungen eines Hardware-Kernels gegenüber einer eingebetteten CPU von bis zu 37-fach. Neben Betrachtungen einzelner Hardware-Kernels untersuchen wir auch die Beschleunigung auf Programmebene. Für eine mehrstufige Bildkomprimierung erreichen wir dabei eine Beschleunigung um den Faktor 5 gegenüber einer superskalaren, eingebetteten CPU mit 400 MHz.

# Acknowledgments

I would like to thank Professor Andreas Koch. He gave direction to my research and guided me throughout the development phase of this work. Furthermore, the readability of the text was significantly improved by considering his comments. Andreas has acquainted me with adaptive computers during my computer science studies.

I would also like to thank Professor Ulrich Golze for his steady support. The comprehensibility of the formal descriptions in this thesis improved notably by taking his suggestions into account. He furthermore provided the working environment, including hardware and software infrastructure.

Several colleagues have provided me with essential building blocks. The preliminary work of Nico Kasprzyk on COMRADE 1.0 is the basis of the COMRADE 2.0 compiler developed in this work. Benjamin Thielmann provided an initial implementation of Modlib and LMEM. Holger Lange and Thorsten Wink provided and adjusted MARC as well as the ACE-M5 design and operating system.

The ML507 development board was donated by Xilinx.

Spelling and grammar in this work were improved by considering Nadia and Michael Acland's comments.

The moral support from my wife Hendrikje Lütjens was invaluable throughout this work.

I thank my parents Edith Gädke-Döblitz and Jürgen Gädke for their moral support as well as the financial support provided during the completion of my computer science degree, laying the groundwork for this thesis.

# Contents

# 1. Introduction

Computers advance into an expanding set of application domains. They are used in commerce and research, in industry as well as in education, entertainment, and infotainment. Many applications require increasingly more computational performance for more accurate simulations, faster data transformation, higher throughput, more realistic games, and ever more features. Mobile applications have to achieve these objectives with a limited power budget. Most of these applications are written in software which is executed on a **central processing unit (CPU)**. Despite numerous sophisticated enhancements, CPUs currently suffer from the 4 GHz performance barrier which seems to be insurmountable without fundamental architectural modifications. A very promising option is the usage of flexible hardware accelerators which can be adjusted to the currently executed application. Such accelerator devices do exist, however, they are not used commonly today due to the lack of adequate compilers. This work develops such a compiler.

During the past 40 years, the CPU complexity (i.e., the number of transistors) has increased exponentially (Fig. 1.1) according to Moore's Law (these numbers relate to Intel only, however, similar results hold for other CPU brands). Similarly, the CPU frequency has increased exponentially, continuously boosting the CPU performance. Unfortunately, this has caused an exponential growth in CPU power consumption, exceeding 100 W in 2003. Due to the resulting heat generation, frequencies higher than 4 GHz are hard to handle, so that we have been experiencing a frequency stagnation between 3 and 4 GHz since then. Instead of increasing the frequency, the CPU manufacturers now integrate multiple CPU cores in a single chip and use ever larger caches, so that the number of transistors per chip still grows exponentially. While multiple cores are a good choice to improve the performance of multi-threaded applications, they do not help to increase the performance of single-thread programs (making up a good portion of the existing SW today). As both the CPU frequency and the number of instructions processed per clock cycle have stagnated, the single-thread performance has also been stagnating since 2003.

Thus it makes sense to investigate alternative compute models. A possible option is **adaptive computers**. These combine a conventional CPU with a **re-**

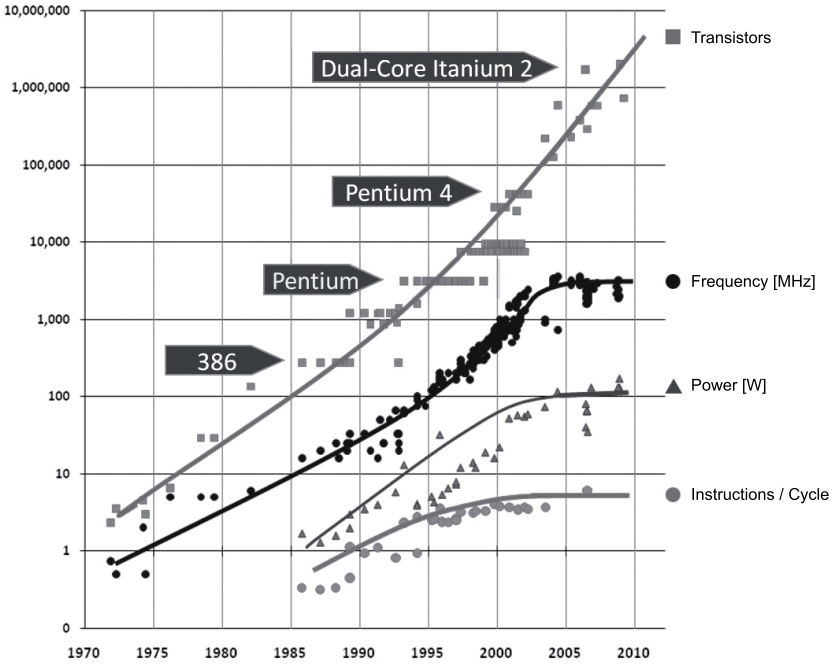


Figure 1.1.: Intel CPU trends [Josh11].

**configurable** device used as hardware (HW) accelerator. Such a device is able to emulate arbitrary hardware circuits[1], and it can be reprogrammed as often as required to fit the needs of a given application. The CPU executes those parts of an application which cannot be accelerated by the reconfigurable device. This computational concept is called **adaptive computing**. The most popular reconfigurable devices used for adaptive computing are **field programmable gate arrays (FPGAs)**, offering a fine-grain and regular structure of reconfigurable units (e.g., look-up tables and flip-flops) as well as hard-wired functional blocks such as memory, multipliers, and even embedded CPU cores.

Until today, adaptive computing has successfully been used to accelerate applications from a wide spectrum of fields. The classic example is the gene sequence comparison. In 1993, the SPLASH-2 ACS [Hoan93] achieved a speed-up of 1,300 over one of the fastest commercial computers available at that time. Currently, the largest speed-up reported in this field is 3,000, achieved by a system consisting of an Intel Xeon CPU @ 2.8 GHz and two Xilinx Virtex-4 FPGAs, over the Xeon alone [BuNa08]. Other worthwhile speed-up examples (yet not that spectacular) can be found in cryptography (10x) [ShVu93], wavelet image compression (7x) [GäKo04], molecular dynamics simulations (10x) [GuHe07], and exhaustive search (27x) [Schu07].

In addition to the speed-up, adaptive computing is promising due to its low energy consumption. Despite speeding up the computation over a general-purpose CPU, an FPGA often reduces power consumption by an order of magnitude [GäKo04] [LSKH09].

Even with these advantages, adaptive computing has not yet become a mainstream methodology. Probably the most prominent reason for this is the programming complexity. Before a program can run on an adaptive computer, it has to be partitioned into software (SW) and hardware (HW) parts, and both parts must be designed according to the target technology, as shown in Fig. 1.2.

Traditionally, HW/SW partitioning is done manually, and the FPGA is programmed using a hardware description language (HDL) such as Verilog or VHDL. Both steps, especially the latter, require special knowledge about hardware design and the target architecture, forming a barrier for the large majority of programmers who are short on HDL experience. Furthermore, designing an application in an HDL is much more costly and time consuming compared to software development. Thus, an important issue for the acceptance of adaptive computing is the development of high-level tools that simplify programming

[1] The complexity of the HW circuit that can be emulated is limited by the area resources provided by the device.

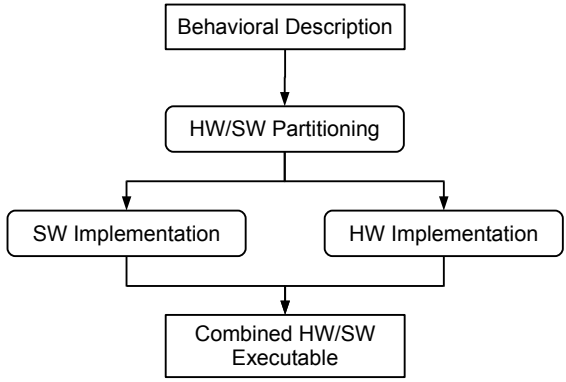


Figure 1.2.: Design flow for adaptive computer applications.

for adaptive computers. At this point, adaptive computing of course overlaps with general **high-level synthesis (HLS)**, which is the "automated generation of the hardware circuit of a digital system from a behavioral description" [GGDN04].

Many of today's high-level tools are built on software compiler frameworks for two reasons. First, the input language (e.g., C) is already widespread and known to many users. Second, many existing code optimizations from the software world can be re-used for hardware development. Example SW frameworks used in HW generation are SUIF2 [ADHL00] (used by COMRADE [KoKa05], ROCCC [GuNB08], and Molen [Pana07]) and LLVM [Latt02] (used by CHiMPS [PBDM08]).

By now, a generic compile flow (or *synthesis flow*) for creating hardware from high-level languages (HLL) has emerged (Fig. 1.3). From the HLL representation, a *control flow graph* (CFG) is built and optimized by several compiler passes. Scalar replacement of array accesses is an important method to reduce memory accesses, which is often critical for efficient hardware solutions. Constant propagation, common subexpression elimination, and dead code elimination reduce the hardware area required. Loop unrolling can increase the *instruction level parallelism* (the number of operations executable in parallel), at the expense of bigger designs. Furthermore, the CFG conversion to static single assignment (SSA) form has proven to be very useful for the transformation to low-level, hardware-centric representations such as *(control) data flow graphs* ((C)DFGs). These are used as intermediate steps before creating the actual hardware representation (typically in an HDL).

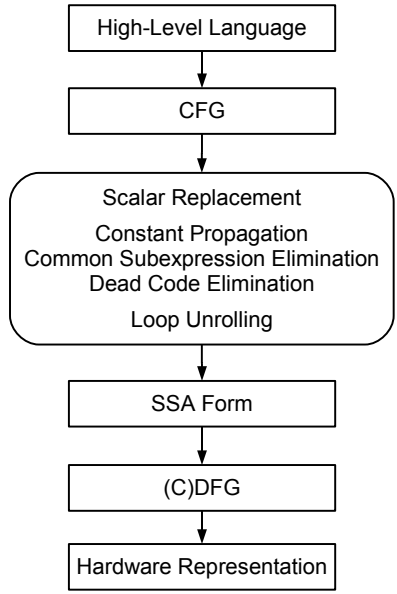


Figure 1.3.: Generic compile flow for hardware generation from high-level languages.

The generation of the hardware representation (as well as several intermediary compile passes) heavily depend on the scheduling technique used. **Scheduling** determines which parts of a hardware computation are executed at a certain time. **Static scheduling** (Fig. 1.4(a)) defines fixed time slots (1, 2, 3 in the Figure) which are assigned HW operations at compile time. A global state machine controls (at runtime) which time slots (and thus HW operations) are currently active.

**Dynamic scheduling** (Fig. 1.4(b)) does not use such time slots. Instead, operations execute independently as soon as their input data is available. The availability of input data is denoted by the presence of tokens (the circles containing a + in the Figure) similar to Petri-nets. Such self-organizing systems are more flexible and better suited for variable operator latencies which can, for example, occur when a cached memory is accessed. While reading a data word from the cache could take just 2 cycles, the same read access might need much longer (e.g., 100 cycles) if that data word has to be loaded from the main memory. If such a memory stall occurs in a statically scheduled system, the complete state machine stalls. In dynamic scheduling only the read operation stalls, while other independent operations can continue their computation. Thus dynamic scheduling can save execution time versus static scheduling.

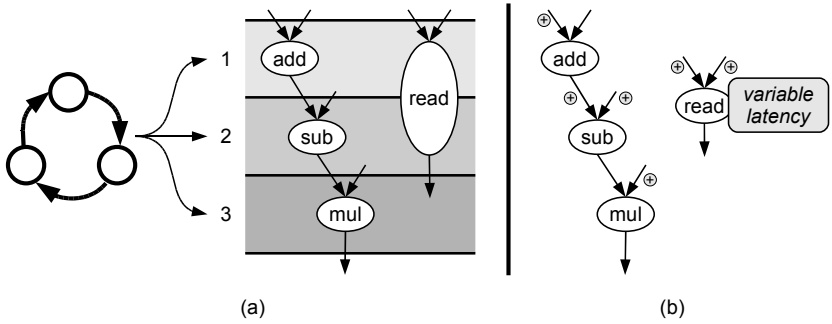


Figure 1.4.: (a) static scheduling, (b) dynamic scheduling.

While static scheduling is used by most current synthesis flows, only limited research has been done on dynamic scheduling in the context of adaptive computing to date. This work examines the applicability of dynamic scheduling to a compile flow for adaptive computers.

Our major goals are:

- The design of a hardware execution model using dynamic scheduling, suited for compilation from high-level languages to hardware descriptions for adaptive computers.
- Simple programmability, i.e., restrictions and extensions of the input language should be kept minimal.
- The evaluation of the execution model using an appropriate compiler framework.

This thesis is structured as follows: Chapter 2 defines basic terms used throughout this work. Chapter 3 introduces the initial version of COMRADE, a high-level compiler for adaptive computers used as a basis for this work. Chapter 4 reviews other existing high-level compilers. Chapter 5 illustrates the HW/SW execution model, which is then formally defined in Chapter 6. Chapter 7 describes the hardware operator library used by our approach. The compile flow which integrates our model is presented in Chapter 8. Chapter 9 gives simulation and synthesis results obtained for different optimization techniques, while Chapter 10 examines the application-level speed-up. The work is summarized in Chapter 11.

# 2. Basics

This Chapter defines fundamental terms used throughout this work.

An **adaptive compiler** is a compiler which transforms a program written in a high-level language (e.g., C, C++, Java) into a combined HW/SW executable for an adaptive computer. An adaptive compiler can use several **intermediate representation**s (**IR**) of the input program to simplify compiler optimizations or to successively transform the input program into the target representation. The two most prominent IRs used in adaptive compilers and high-level synthesis are control flow graphs and data flow graphs; these are discussed in the following two Sections. Subsequently, Section 2.3 introduces important terms related to the scheduling of hardware operations.

## 2.1. Control Flow

**Definition 2.1.1.** A **control flow frame** $CF = (N, E, ann, \mathsf{start}, \mathsf{end})$ consists of

- a finite directed graph $(N, E)$ with nodes $N$ and directed edges $E \subseteq N \times N$,
- an annotation $ann : E \to \mathbb{Z} \cup \{\epsilon\}$ assigning an integer or $\epsilon$ to each edge,
- and two distinct nodes $\mathsf{start}, \mathsf{end} \in N$.

Furthermore, every node is reachable from start, and end is reachable from every node. When a control flow frame is used to model the control flow of a program, *ann* annotes the outgoing edges of conditional branches (e.g., if, switch in C) to define a branch target depending on the outcome of the condition. The $\epsilon$ value is used for default successors in the switch case as well as for ordinary (non-branching) edges.

Fig. 2.1(a) shows a sample *CF*: Edges $(a, b)$ and $(a, c)$ are annotated with 0 and 1 respectively, while the remaining edges are annotated with $\epsilon$, which is usually omitted in graphical representations.

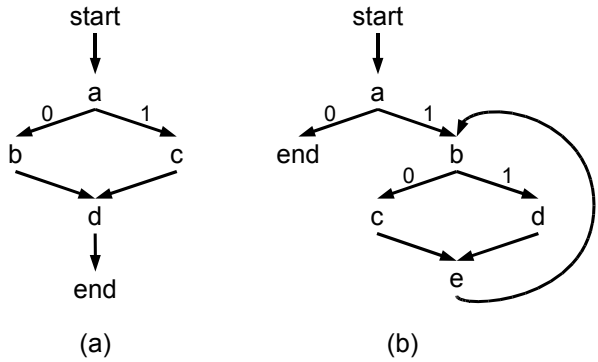


Figure 2.1.: Graphical representations of control flow frame samples.

**Definition 2.1.2.** A **basic block** is a (possibly empty) sequence of program statements without intermediary branches, i.e., at most the last statement of the sequence may be a branch statement.

We now combine the control flow frame with basic blocks to define the control flow graph.

**Definition 2.1.3.** A **control flow graph** $CFG = (CF, BB, nb)$ consists of

- a control flow frame $CF$,
- a set of basic blocks $BB$, and
- an assignment $nb : N \to BB$ of nodes to basic blocks.

A $CFG$ represents a program by mapping each node of a control flow frame $CF$ to a basic block, the edges of $CF$ defining the program order. As anticipated in the $CF$ definition, the edge annotations define which of the succeeding basic blocks is executed after a conditional branch. In a graphical representation, the statements contained in a basic block $b$ are typically printed inside $CFG$ node $n$, if $n$ is mapped to $b$, i.e., $nb(n) = b$.

**Definition 2.1.4.** Given a $CFG$ with nodes $N$, edges $E$, start and end node, we define several useful terms:

- $n \in N$ is a **branch node**, if $n$ has more than 1 successor.

- $n \in N$ is a **join node**, if $n$ has more than 1 predecessor.
- A **region** is a set of nodes $R \subseteq N$.
- A **path** $p$ is a list of nodes, such that for each two successive nodes $n$, $m$ in $p$, there is an edge $(n, m) \in E$. If the first node of $p$ is $n$ and the last node is $m$, then $p$ is said to be a **path from** $n$ **to** $m$.
- $d \in N$ **dominates** $n \in N$, if every path from start to $n$ contains $d$.
- $z \in N$ **post-dominates** $n \in N$, if all paths from $n$ to end contain $z$.
- The **post-dominance frontier (PDF)** of $z \in N$ is a subset $\{n_1, \ldots, n_k\}$ of $N$, such that for $n_i \in PDF(z)$: $n_i$ is *not* post-dominated by $z$, and $n_i$ has a successor which *is* post-dominated by $z$.

Referring to Fig. 2.1(b), we give some examples for the latter definitions: $a$ and $b$ are branch nodes, $e$ is a join node, $\{a, b, c, e\}$ is a region, and $(a, b, d)$ is a path. $a$ dominates $b$, $c$, $d$, $e$, and end. $e$ post-dominates $b$, $c$, and $d$. The post-dominance frontier of $e$ is $\{a\}$.

The post-dominance frontier is essential for finding the CFG nodes which determine if a given CFG node is entered during the program execution. We point this out with the notion of control dependence.

**Definition 2.1.5.** Given a *CF* with node set $N$ and nodes $c, n \in N$, $n$ is **control dependent on** $c$ if $c$ is contained in the post-dominance frontier of $n$.

If $n$ is control dependent on $c$, we call $c$ a **controller of** $n$, or in short: $c$ **controls** $n$. Note that in contrast to Ferrante's original definition of control dependence [FeOW87], nodes never control themselves, which is a consequence of the post-dominance frontier definition above.

To give an example, in the *CF* in Fig. 2.1(b), $a$ controls $b$ and $e$, while $b$ controls $c$ and $d$. Note that $a$ does *not* control $c$ or $d$, because neither of the two post-dominates $b$.

In high-level synthesis (HLS), it is often desirable to simplify the control flow of a program. This simplifies optimization passes on the one hand, and can be a requirement for certain hardware implementations on the other. In this work, structuredness as a central property of programs and *CFG*s is a requirement for hardware generation. A **structured program** uses only concatenation, selection, and repetition of statements as defined by Dijkstra[1] [Dijk72]. In particu-

[1] Dijkstra disapproves of goto statements explicitly and focuses on concatenation, selection, and repetition. However, the principle of using only these three control constructs is also known as imperative programming and has been used long before Dijkstra's definition.

lar, a program which does not contain any goto statements, neither explicit nor implicit (e.g., break and continue statements in C loops) is structured. The structured programming theorem [BöJa66] states that every computable function can be implemented in a programming language that combines subprograms using only these three control structures. Thus, we can restrict HLS to structured programs without loss of generality: unstructured programs can be made structured before the actual HLS is applied.

Beyond structuring, we can further normalize programs prior to the HLS. Bottom-testing loops (e.g., do-while in C) can be replaced by top-testing loops (while, for) by inserting an additional flag into the loop condition as shown in Fig. 2.2. We call a structured program without bottom-testing loops a **top-structured program**, or short **t-structured program**.

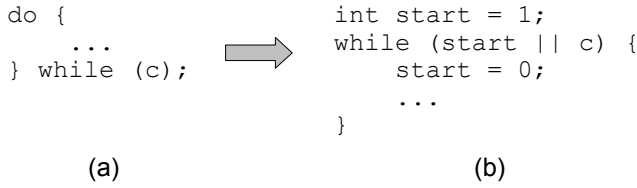


Figure 2.2.: (a) Bottom-testing loop; (b) replaced by top-testing loop.

According to t-structured programs, we define t-structured *CF*s in a bottom-up manner, using Dijkstra's three building blocks of structuredness.

**Definition 2.1.6.** A **t-structured control flow frame** is a control flow frame which can be built up starting with the initial *CF* illustrated in Fig. 2.3(a) and successively applying one of the three **t-structured transformations** (b), (c), (d) shown in Fig. 2.3.

The example in Fig. 2.4(a) shows a t-structured *CF*. It can be generated by applying the substitutions in the order (b), (c), (d) to the initial *CF*. In contrast, the *CF* shown in Fig. 2.4(b) is not t-structured, which can be proven by trying all combinations of at most four substitution steps (every substitution adds at least one node). As an indication for the non-t-structuredness of this example, observe that after generating the alternative branches $(a, b)$, $(a, c)$ using substitution (c), there is no substitution which can insert an edge from $c$ to $b$.

The t-structuredness can of course be propagated to *CFG*s.

**Definition 2.1.7.** A **t-structured control flow graph** is a control flow graph

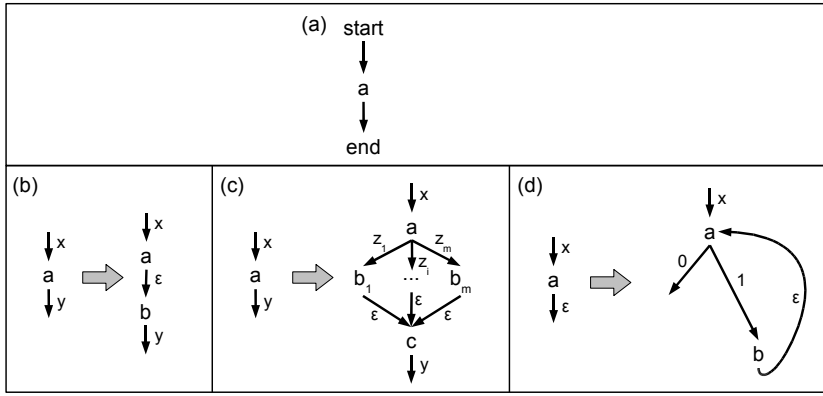


Figure 2.3.: (a) Initial *CF*; (b) concatenation; (c) selection; (d) repetition.

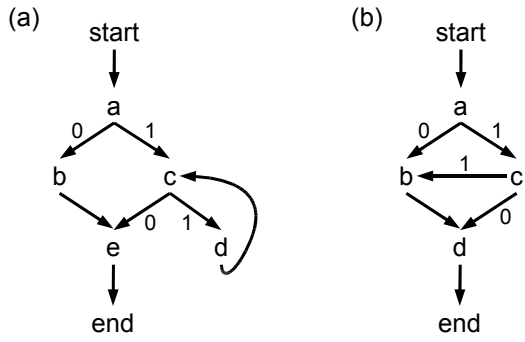


Figure 2.4.: (a) T-structured *CF*; (b) non-t-structured *CF*.

$CFG = (CF, BB, nb)$, such that its control flow frame $CF$ is t-structured.

**Lemma 2.1.1.** *Every t-structured program can be represented by a t-structured CFG.*

*Proof.* This can be shown by induction over the types of statements allowed in the source language, which we omit here for brevity. □

In Fig. 2.3, transformations (c) and (d) introduce new branch and join nodes. Note that this induces a one-to-one mapping $\rho$ from the inserted branch nodes to the inserted join nodes: $\rho(b) = j$ maps a branch node $b$ to the join node $j$ which has been introduced during the same transformation in which $b$ has been introduced. Note that a branch node introduced by (d) is mapped to itself, being a join node at the same time.

**Proposition 2.1.1.** *In a t-structured CFG, every node has at most one controller.*

*Proof.* We show this by induction over the t-structured transformations. In the initial control flow frame (Fig. 2.3(a)), obviously none of the three nodes has a controller. We now assume that none of the nodes in a given t-structured *CFG* has more than one controller. We pick an arbitrary node $a$ and apply one of the t-structured transformations shown in Fig. 2.3(b)-(d). First, we assume that concatenation (b) is applied. If $a$ had a controller before applying the transformation, the inserted node $b$ has the same controller after the transformation, because $b$ post-dominates $a$. Otherwise, neither $a$ nor $b$ have a controller. Second, we assume that selection (c) is applied. If $a$ had a controller before the transformation, then $a$ and $c$ obviously have the same controller after the transformation (and no controller otherwise). The newly inserted alternative nodes $b_1, \ldots, b_m$ are controlled by $a$ (and only $a$), because they do not post-dominate $a$. Third, we assume that repetition (d) is applied. Because $b$ does not control $a$ and $a$ does not control itself, the transformation leaves the control dependence of $a$ invariant. The newly inserted node $b$ is control dependent on $a$ (and only on $a$), because it does not post-dominate $a$. □

The next definitions give a notion of loops, consisting of loop header and loop body.

**Definition 2.1.8.** Given a t-structured *CFG* with node set $N$ and edge set $E$.

- $e = (n, d) \in E$ is a **back-edge**, if $d$ dominates $n$.

- $l \in N$ is a **loop header**, if it has an incoming back-edge.

Due to the structuredness, a loop header has exactly one incoming back-edge. Given a loop header $l$ with incoming back-edge $(n, l)$,

- the **loop body** $LB(l) \subset N$ **of** $l$ contains all nodes from which $l$ is reachable only by passing through $n$, i.e., $LB(l) = \{k \in N : \forall \text{ paths } p \text{ from } k \text{ to } l : n \in p\}$.

Examples for the latter definitions can be found in Fig. 2.1(b): $(e, b)$ is a back-edge, $b$ is a loop header, and $\{c, d, e\}$ is the loop body of loop header $b$.

We conclude this Section by introducing several terms needed for hardware/-software partitioning *CFG*s.

**Definition 2.1.9.** A **partitioned control flow graph** consists of

- a control flow graph with node set $N$, and
- an additional mapping function $m : N \to \{hw, sw, swsub\}$, which maps each node to either hardware (hw), software (sw), or sub-software (swsub). Swsub nodes will be used to model SW services (see below).

**Definition 2.1.10.** Given a partitioned *CFG* $(N, E, ann, \mathsf{start}, \mathsf{end}, m)$.

- A **hardware region** (HW region) is a region in which all nodes are mapped to hardware or sub-software. Furthermore, a HW region $H$ is maximal in the sense that no node in $H$ has a successor or predecessor node which is mapped to hardware, but which is not contained in $H$.
- A **software region** (SW region) is a region in which all nodes are mapped to software. A SW region is maximal in analogy to HW regions.
- A **software subregion** (SW subregion) of a hardware region $H$ is a weakly connected subset $S$ of $H$, such that all nodes of $S$ are mapped to sub-software and such that $S$ is maximal in analogy to HW regions.
- A **hardware-to-software transition** (HW/SW transition) is an edge $(h_1, s_2) \in E$ such that $m(h_1) = hw$, and $m(s_2) \in \{sw, swsub\}$.
- A **software-to-hardware transition** (SW/HW transition) is an edge $(s_1, h_2) \in E$ such that $m(s_1) \in \{sw, swsub\}$, and $m(h_2) = hw$.

## 2.2. Data Flow

While *CFG*s, based on sequential statements in basic blocks, form a software-related intermediate representation (IR), data flow-based representations are more hardware-centric and thus belong to the low-level IRs of adaptive compilers. To give a definition of the popular data flow graph, we adhere to [GGDN04].

**Definition 2.2.1.** A **data flow graph** (*DFG*) consists of

- a finite, directed, acyclic graph $(N, E)$,
- a set of operations $OP$, and
- an annotation $op : N \to OP$ assigning each node an operation.

Each instance of an operation assigned to a node can produce and consume data. The edges $E \subseteq N \times N$ represent data flow dependences, i.e., a directed edge $(a, b)$ exists in $E$ if data is produced by $op(a)$ and consumed by $op(b)$. For simplicity, we will write $a$ instead of $op(a)$ when addressing the operation assigned to $a$.

Fig. 2.5(b) shows the data flow graph for the code shown in Fig. 2.5(a). Note that the variable identifiers shown in Fig. (b) are not part of the *DFG*. The same applies for the edges lacking a source or target node; they are shown just to clarify the correlation to the code in Fig. (a).

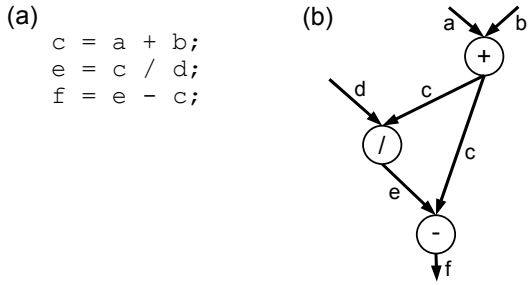


Figure 2.5.: (a) Sample C code; (b) *DFG* representing the C code.

*DFG*s can be used to model synchronous hardware on the register transfer level. In this case, the behavior of each operation is aligned to a dedicated

clock signal. The time between two rising clock edges is one **cycle**. In this context, we define several properties of operations.

**Definition 2.2.2.** Given an operation $p$. The **latency** of $p$ is a non-negative integer denoting the number of cycles needed to compute a result from the operation input values.

**Definition 2.2.3.** Given an operation $p$ with latency $lat$. The **initiation interval** $ii$ of $p$, $0 \leq ii \leq lat$ denotes the mininum number of cycles we have to wait before we can feed the next set of input values into $p$. Note that the latency can be 0, which allows us to model the execution of several consecutive operations during the same cycle.

**Definition 2.2.4.** Given an operation $p$ with latency $lat > 1$ and initiation interval $ii$. If $ii < lat$, then $p$ is **pipelinable**. If $ii = 1$, $p$ is **fully pipelinable**. If $ii = lat$, $p$ is **non-pipelinable**.

**Definition 2.2.5.** A **timed data flow graph** consists of

- a data flow graph,
- a **latency function** $lat : OP \rightarrow \mathbb{N}_0$, assigning each operation a latency, and
- an **initiation interval function** $ii : OP \rightarrow \mathbb{N}_0$, assigning each operation an initiation interval.

*DFG*s and timed *DFG*s are limited to data dependences. For example, they do not account for control dependences such as if/else structures in high-level languages (Fig. 2.6(a)). To a certain extent, it can be reasonable to replace control dependences with pure data flow (Fig. 2.6(b)). A disadvantage of this is that every operation is executed, even if their results are not needed at runtime, e.g., when they originate from a branch which is not taken. Depending on the scheduling strategy, this can have a negative impact on the execution time and the energy consumption. Furthermore, eliminating the control dependences of operations with side effects (e.g., write accesses to the memory as shown in Fig. 2.6(c)) is impragmatic in most cases, because this would either require side effect fixes or a redesign of the source code in order to replace side effect operations by operations without side effects (e.g., completely removing pointers and arrays).

Control dependences can be integrated into the *DFG* concept by extending the *DFG* with control (dependence) edges. The resulting structure is called **control**

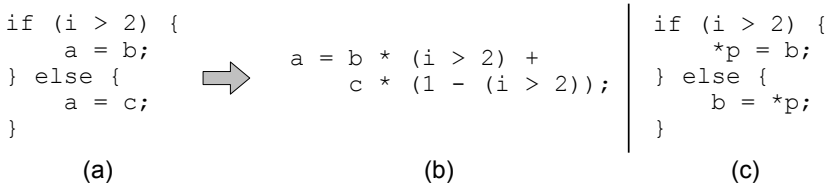


Figure 2.6.: (a) Sample C code with control dependences; (b) control dependences replaced by data dependences; (c) Sample C code containing statements with side effects (memory accesses).

**data flow graph** (*CDFG*)[2]. Control edges are annotated with conditions, their result depending on the result of the source node operation. Typically, control edges represent the **predicated execution** semantics, i.e., the target node is executed only if the control edge condition is true. Fig. 2.7 shows the *CDFG* for the C code in Fig. 2.6(c). The two control edges represent the if/else control structure: If $i > 2$ is true (==1), the store is executed, otherwise the result is false (==0) so that the load is executed.

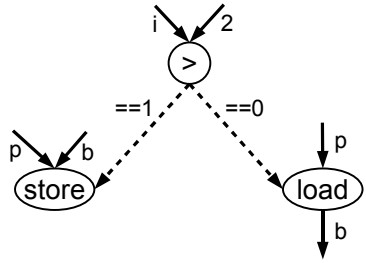


Figure 2.7.: *CDFG* for the C code in Fig. 2.6(c).

To save runtime, it can be reasonable to execute control dependent operations (i.e., targets of control edges) already before it is known if the result is actually needed. This technique is called **speculative execution**. Speculative execution of an operation is of course only allowed if the operation has no side effects or if there are mechanisms to care for the impacts of side effects (e.g., a memory system allowing speculative write accesses). Practically, speculative execution of an operation $n$ can be implemented in two ways. The first way is to re-

[2]This is only a first glance at CDFGs; we will later define a model in detail which additionally contains memory dependences.

move the control edge(s) that target(s) *n*. The second way is to change the control edge semantics so that *n* is executed as soon as its data dependences are fulfilled, but the result is not propagated to successive operations before the control dependences are fulfilled. We term the latter technique **speculative predicated execution**.

## 2.3. Hardware Operation Scheduling

When we transform a data flow-oriented representation to an actual hardware representation (e.g., register transfer logic in a hardware description language) we have to define a **schedule**, i.e., we define *when* each of the operations is executed. A well-known and widely used technique is **static scheduling**, which schedules the operations into fixed time slots. This approach keeps the control logic simple (ordinary state machines), and it allows **resource sharing**, i.e., using the same pieces of hardware to compute different operations during different time slots. However, static scheduling requires the compiler to know the latencies of each operation. If an operation has a variable latency (e.g., a cached memory access), the compiler schedules for an expected latency. If the actual latency is smaller than expected, successive operations execute later than they could. If the actual latency is greater than expected, the complete state machine has to be stalled. Both cases result in idle time. Another disadvantage is that the number of registers or pipeline steps in a computational path of several operations is fixed. It is not possible to locally insert or remove a register without adjusting the whole state machine. The insertion of registers can be useful to enable higher frequencies; register removal in turn reduces latencies, but increases combinatorial logic delays.

**Dynamic scheduling** is a more flexible, yet less common technique. Instead of mapping operations to fixed time frames, it defines conditions which determine the time frames at runtime. Thus, even operations with variable latencies can be handled adequately. For this flexibility, dynamic scheduling accepts an area increase of the resulting hardware. First, the **sequencer**, i.e., the logic determining when to enable which operation, is larger than a state machine which would have been used for static scheduling. Second, there are fewer resource sharing opportunities because it is not known at compile time when an operation will actually execute. In dynamic scheduling valid data items are marked by tokens, usually implemented as a single bit of information at data inputs and outputs. More sophisticated models (such as this work) use different kinds of tokens (cf. Section 4.2).

Dynamically scheduled HW can be viewed as a **latency-insensitive system (LIS)** [CaMS01]. In an LIS, composed of modules and communication channels between the modules, the channel latency does not affect the correctness of the system. Similarly, the latency of an operator in dynamically scheduled HW does not affect the correctness of the HW.

# 3. COMRADE 1.0

The roots this work has emerged from reach back to the NIMBLE compiler [MacM01]. The aim of NIMBLE was to synthesize hardware from ANSI C input code. The drawbacks of NIMBLE included the restriction of hardware creation from innermost loops only, and its technology dependence. Only Xilinx XC4000 FPGAs as well as the GARP architecture [HaWa97] were supported as target platforms.

To overcome these restrictions, Nico Kasprzyk tried to generalize the hardware creation techniques used by NIMBLE. He developed a new compiler for adaptive computers from scratch between 2001 and 2005 [Kasp05]. The result of this work was the first version of the COMRADE compiler, which we therefore call COMRADE 1.0. This Chapter gives an overview of the compile flow and implementation. We will also point out limitations (both conceptual and concerning the implementation) to better highlight the progress made in COMRADE 2.0, one of the main achievements of this current work. Note that despite its weakness, COMRADE 1.0 was a far more ambitious project than NIMBLE and provided numerous insights useful for guiding our own research.

## 3.1. Features

The basic idea of COMRADE is to support the complete ANSI C language without additional annotations, including arbitrarily nested loops and control conditions as well as arbitrary array references and pointers. Furthermore, hardware creation should not be bound to a specific target FPGA to support virtually any reconfigurable compute platform.

When creating hardware in such a general way, using an advanced computational model is essential for the hardware efficiency (e.g., concerning the runtime) compared to competitive computing architectures, such as execution on an embedded CPU. Therefore, COMRADE has been designed to generate dynamically scheduled, speculative hardware from the beginning. Dynamic scheduling allows efficient execution even in the presence of variable oper-

ator latencies. Speculative execution is another technique used to decrease the runtime: Before a branch occurs, the computations in all branch targets are already precomputed (if this is not limited by other dependences). This technique is very amenable to hardware, because it can be done without side effects such as pipeline refills after a branch misprediction in CPUs.

## 3.2. Compile Flow

The COMRADE 1.0 Compile Flow is depicted in Fig. 3.1. COMRADE is based on the Stanford SUIF2 framework [ADHL00] and therefore consists of several SUIF2 passes. Inputs to COMRADE are a C program in source code, user constraints, and the GLACE module generators [NeKo01]. After an initial preparation, which includes a C preprocessor (CPP) run on the input C code and the transformation to the abstract syntax tree (AST) based SUIF2 format, COMRADE performs its own analysis and transformations. As a first step, the dynamic profiling pass compiles the program using a standard software compiler, executes it on a sample input data set (specified in the user constraints), and attaches the gathered profiling data to the AST statements. The next pass inlines the procedures which are called most frequently; the inlining threshold is obtained from the user constraints. This elimination of procedure calls simplifies the later hardware generation. After that, a control flow graph (CFG) is built for each procedure. This includes the execution frequency annotation of each CFG node.

In the HW/SW partitioning pass, the CFG nodes exceeding a user defined execution frequency are marked for hardware execution, while the other nodes remain software-associated. Connected hardware CFG nodes form a hardware region in the CFG, called a **hardware kernel** after the ensuing transformation to hardware. By construction, a hardware region is a (possibly nested) loop. For the case of a nested loop, Kasprzyk argues that there might be cases in which it cannot be determined at compile time whether execution of the whole loop nest or only of the deeper nesting levels is more efficient. He has therefore integrated the loop duplication pass, which copies a hardware-associated loop nest multiple times and thus generates all reasonable HW/SW associations of the nesting levels. This allows for switching between hardware and software execution on runtime conditions.

The next pass performs an array iteration space analysis using the Omega framework [Pugh91]. Results of this test can be used for later optimization passes.

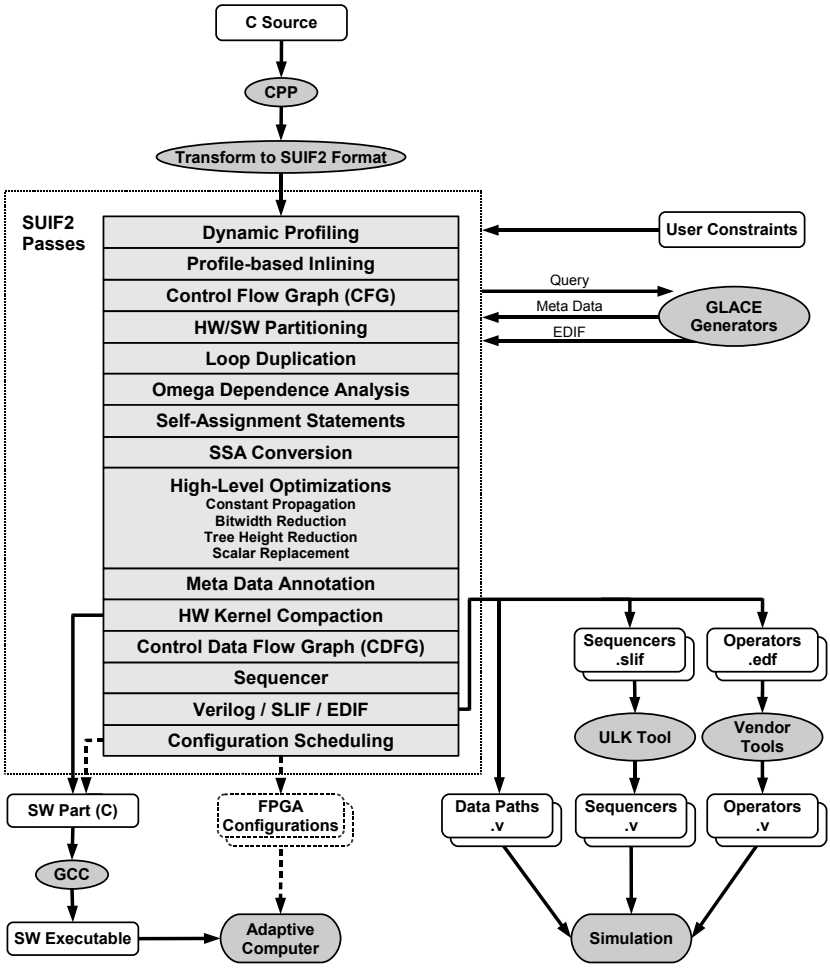


Figure 3.1.: The COMRADE 1.0 Compile Flow.

Before the conversion to the SSA form [CFRW91], COMRADE iterates over all loop bodies in HW regions and inserts a self-assignment statement (such as `n = n;`) for each variable which is read by an expression, but not assigned any value in the current loop body. Thus, after SSA conversion, for every variable that is written-to or read-from, there is a phi statement in the loop header having two inputs: one from outside the loop, and a feed-back input from the loop body itself. This feed-back ensures that each variable is reactivated for each loop iteration.

A basic concept in COMRADE 1.0 is the correspondence between C operations as well as statements in the CFG and the HW operators modeled later in the CDFG. Therefore, COMRADE inserts dedicated statements at HW/SW and SW/HW transitions to model the variable transfer between software and hardware. At HW/SW transitions, a statement for the sending of an IRQ signal is inserted. These statement insertions are performed after the SSA conversion, but integrated into the SSA conversion pass.

Next, several high-level optimizations improve the hardware eligibility, among them constant propagation, bitwidth reduction, height reduction of abstract syntax trees, and scalar replacement. The latter uses dependence information gained from the previous Omega analysis to forward data loaded from memory to successive loop iterations as well as to relocate array accesses, reducing the total number of memory accesses.

The C expressions and statements in the CFG are then annotated with meta data describing the area requirements and the critical path of their hardware operator counterparts. This meta data is obtained from the GLACE module generator library.

During HW kernel compaction, hardware region CFG blocks are reverted to software if they do not fit onto the target FPGA or contain statements which cannot be transformed to hardware. These are floating point computations and (non-inlined) procedure calls, e.g., to the C library. HW/SW interfaces (memory-mapped transfer of live variables, interrupt handling) are inserted accordingly.

Subsequently, a control data flow graph (CDFG) is created for each hardware region. This involves creating hardware operator nodes for each C expression and statement, connecting these nodes by data edges according to the data dependences, and adding further control edges needed for predicated and speculative execution. COMRADE here even inserts memory edges resembling memory dependences, allowing the integration of memory access nodes into the CDFG. The mechanisms used here are still limited and error-prone, but

they form the cornerstone of the more sophisticated COCOMA representation developed in this work (cf. Chapter 6).

Each CDFG is then equipped with sequencer logic used to control the token flow at runtime. Then, COMRADE outputs for each CDFG a data path Verilog module which instantiates and connects the hardware operators obtained from GLACE in the EDIF format [Kahn95] (FPGA vendor tools are used to convert them to Verilog simulation files). COMRADE outputs the sequencers in SLIF format [Drey94], which is converted to Verilog using the conversion tool ULK [Drey94]. SLIF was chosen as the intermediary format as it was able to apply (at that time) advanced logic optimization techniques using SIS [SSLM92].

The final reconfiguration scheduling step [KaVK05] aims to merge hardware kernels to configurations for the adaptive computer, and to schedule the dynamic reconfigurations to be performed during the runtime of the application. The C code including HW/SW interfaces as augmented by the HW kernel compaction pass is compiled with a standard GCC cross compiler and can then be executed on the adaptive computer.

While the configuration scheduling pass determines which kernels should be merged and generates a configuration schedule, this information is not yet used in COMRADE 1.0 to actually create the FPGA configurations and to augment the software with reconfiguration commands. The associated arrows in Fig. 3.1 are therefore printed in dashed style. However, a simulation environment exists which is able to simulate single hardware kernels consisting of a data path, a sequencer and operator module instances.

## 3.3. Deficiencies

This Section describes the major deficiencies and problems of COMRADE 1.0 concerning hardware generation, the front-end, memory accesses, and the overall toolchain.

### 3.3.1. Hardware Generation

For the insertion of control edges into the CDFG, Kasprzyk describes a method using dominator relations on the line graph of the CFG, which he terms **ECFG**. He describes that method only for the insertion of control edges to multiplexer predecessors, lacking support for control hierarchies and memory accesses. However, the actual COMRADE 1.0 source code already contains extensions

of Kasprzyk's initial approach; an example is illustrated in Fig. 3.2.

We assume that CFG node 5 in Fig. 3.2(a) contains a storing memory access, i.e., a write access to an array or pointer. Such a memory access appears as store node in the CDFG. Each store node needs a control edge pointing to it to prevent mis-speculated writes to the memory. To find the origin of that edge in the CDFG, we need to locate the corresponding branch node in the CFG, called the **CFG controller node** of a given CFG node. In Fig. 3.2(a) the CFG controller node of CFG node 5 clearly is 1: If the transition $(1, 2)$ occurs, all statements in node 5 *are* executed; if $(1, 6)$ occurs instead, they are *not* executed.

To find the CFG controller node of node 5, Kasprzyk first picks a CFG edge entering the node. In Fig. 3.2(a) this can be $(3, 5)$ or $(4, 5)$. Without loss of generality, we assume that $(3, 5)$ is picked, corresponding to node 3/5 in Fig. 3.2(b). Next, the immediate dominator of 3/5 is picked, which is 2/3 as shown by the dominator graph in Fig. 3.2(c). For the ECFG node 2/3, Kasprzyk's method then locates the corresponding CFG edge $(2, 3)$ and declares the source node 2 as CFG controller node. If we assume that $(4, 5)$ is picked instead of $(3, 5)$, the algorithm would compute the same result, which is obviously wrong, because we have already seen that node 1 is the node we are looking for. This shows that Kasprzyk's method does not completely solve the control edge insertion problem.

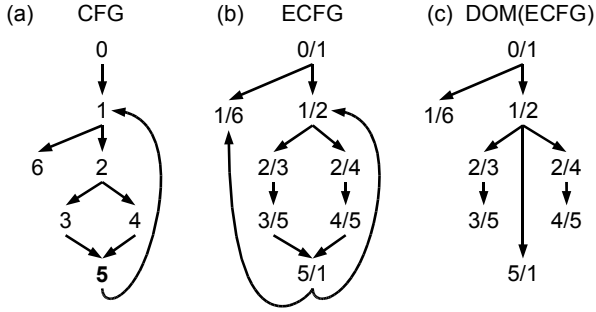


Figure 3.2.: (a) Sample CFG, (b) associated ECFG, (c) ECFG dominators.

Moreover, the token flow in hardware generated from complex control structures has not been sufficiently covered by Kasprzyk's work. Although he mentions cancel token forwarding for hierarchical conditions, he does not address the problem of redundant control dependences [GäKo08]. In general, nested loops produce incorrect hardware, partially due to excess activate tokens in

nodes representing constants. These problems can result in deadlocks or even wrong results when simulating the generated hardware.

Kasprzyk proposes an extension of speculative execution which uses cancel tokens to decrease the runtime of hardware kernels containing imbalanced alternative data paths. Lacking quantitative experimental results, he fortifies his arguments with the example shown in Fig. 3.3. The Figure compares three speculative execution models for alternative branches. In the **late evaluation** model (a), the select node waits until all inputs are available before firing. Using **early evaluation** (b), it fires as soon as the chosen input (the one on the right in the example) is available. Counters at each input of the select node are used to detect mis-speculated results, which can then be discarded. The moving cancel token method (c) proposed by Kasprzyk produces a cancel token instead of increasing a counter. The conceptual difference is that the cancel token moves backwards along data edges and actively cancels a running mis-speculated computation rather than waiting for the mis-speculated result to reach the select node. In Fig. 3.3(c) the second result obviously reaches the select node one cycle earlier than in Fig. 3.3(b). Kasprzyk concludes that moving cancel tokens can save execution time when compared to early execution with input counters. Here, he implicitly assumes non-pipelined operators. If the division operator in the left data path was pipelined, the second division could be started one step earlier (i.e, in step 3) and the second result would then arrive at the select node in step 4, resulting in the same runtime achieved by moving cancel tokens.

The effects of goto, break, and continue statements on the token flow have not been considered. Tests have shown that the computation model used by COMRADE 1.0 is not suitable to directly support the unstructured control flow that these statements can evoke.

In the implementation, the canceling of speculatively executed operations is incorrect. A running operation does correctly propagate a cancel token to its predecessors, but the operation *itself* is not canceled, i.e., as soon as the operation finishes, it emits activate tokens, although it should already have been canceled, resulting in wrong results or deadlocks.

Target nodes of control edges are never executed speculatively. This can have a negative impact on the runtime, when a high-latency operator (such as a divider) is a control edge target; the operation will not start until the control condition has been evaluated.

Furthermore, neither the GLACE operators nor the sequencer support operator-internal pipelining. In the presence of high-latency operators, this results in

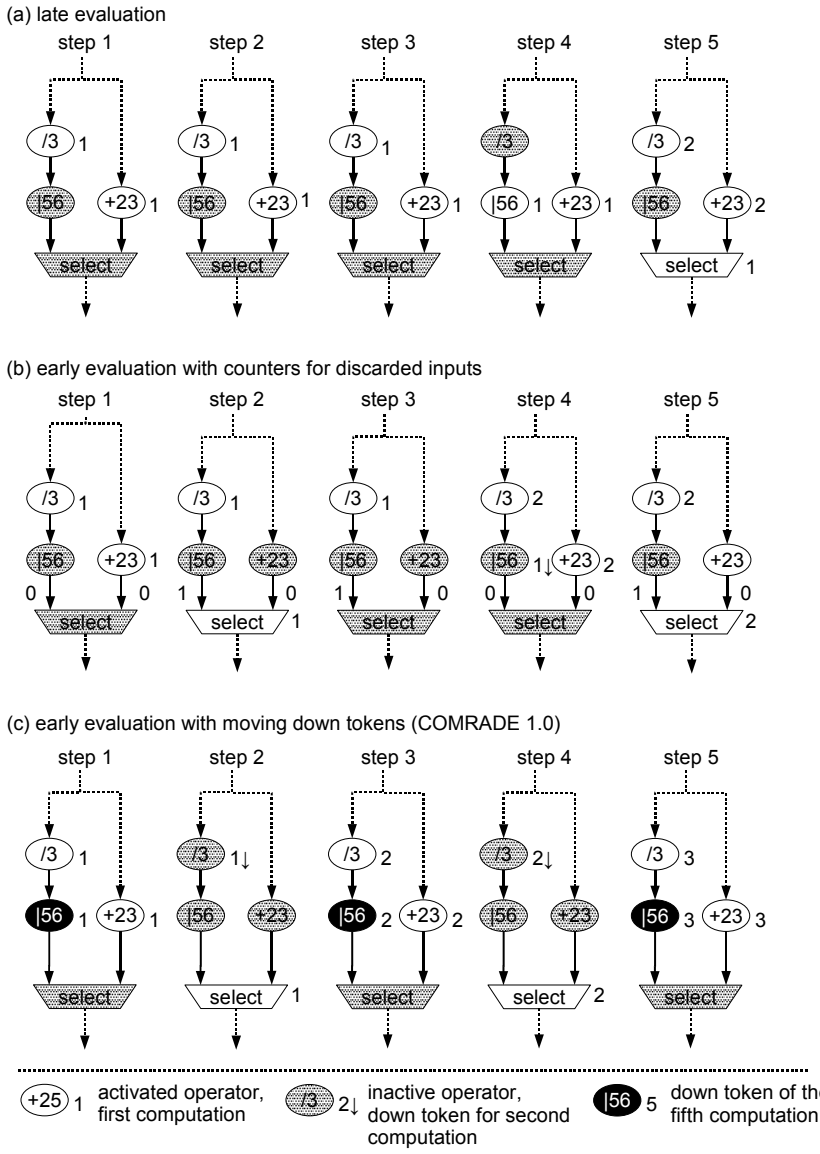


Figure 3.3.: Speculative execution models: Fig. 4.11 from [Kasp05]. Each of the multiplexers selects the right-hand data input.

increased runtimes due to the larger initiation intervals (reduced throughput) of the operators. But even in the absence of such operators in a hardware kernel, the runtime increases due to bubbles in the data paths. An operator holding a result cannot be reactivated before that result has been consumed by all successors. Thus, even a single-cycle operator has an initiation interval of two.

Despite their wide applicability, the GLACE module generators do not support multiplexers with more than two inputs. Therefore, in order to implement bigger multiplexers (e.g., needed for hardware generated from switch statements), Kasprzyk has arranged 2-input multiplexers in a tree structure. The implementation raises two issues. First, the connection of the global select signal to the selects for the 2-input multiplexers is wrong. Second, the input data traverses all multiplexer instances in the tree in one cycle, which limits the achievable clock frequency.

The GLACE library has not undergone a major update since 2004, thus the target technologies it supports (Xilinx XC4000 and Virtex) are obsolete in comparison to current standards. The incompatibility with newer FPGAs is caused by the technology dependent preplacement applied in GLACE. This problem can be bypassed by eliminating such placement directives (generally RLOC constraints) from the generated EDIF netlists. This would, however, result in wrong meta data about area and critical paths, confounding the decisions made during the configuration scheduling pass.

### 3.3.2. Front-end

In addition to the hardware generation issues, other problems are related to the compiler front-end. The loop duplication pass generates all reasonable HW/SW partitions for nested loops on the one hand, but does not provide the software with the intelligence to determine whether it would actually be better to choose the hardware or the software version of a certain loop level. Instead, the hardware version is preferentially used. The remaining alternative hardware regions dedicated to mixed HW/SW execution of nested loops are processed by all successive compiler passes, although they are never really used.

### 3.3.3. Memory Accesses

A hardware kernel generated by COMRADE 1.0 is destined to be connected to an instance of the configurable memory access system MARC [LaKo00]. MARC offers the kernel access to the main memory via a cache port (accesses are cached inside MARC), while accesses to the kernel from the CPU are routed via a slave port (implementing a simple slave interface). Fig. 3.4 illustrates this for the ACE-V [Koch04], which implements an adaptive computer on a circuit board linking a microSPARC-IIep CPU with a Xilinx Virtex 1000 FPGA, the CPU containing a memory controller that allows both the CPU and the FPGA to access the DRAM main memory.

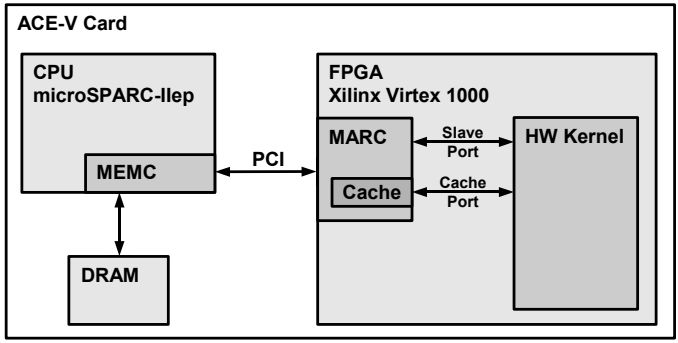


Figure 3.4.: The adaptive computer ACE-V (simplified), used as COMRADE 1.0 target platform.

The COMRADE 1.0 execution model for generated hardware uses a single cache port to a centralized cache. Completely ignoring scenarios with local scratch pad memories as well as multiple access ports to the cache, this restricts the memory bandwidth to 32 bits per cycle. As a result, memory-intensive applications execute more sequentially and less in parallel, with a negative impact on the kernel performance.

The hardware implementation of the memory access operators assume single-cycle latency accesses to the cache, which is often not realistic when a high target clock frequency is required on the target platform.

The insertion of memory edges into the CDFG in order to sequentialize multiple array and pointer accesses for one cache port is rather rudimentary, causing problems in several cases. In an if/else structure containing a memory access only in the then part (not in the else part), a deadlock occurs as soon as the else

part is entered, because consecutive memory accesses (after the if/else) wait for a non-existing activate token. Furthermore, data independent loops and nested loops execute in parallel without obeying memory dependences and without sequentializing their accesses, which can produce erroneous results and deadlocks.

### 3.3.4. Toolchain

In COMRADE 1.0, the compile flow is not completely implemented yet. The compiler outputs several generated hardware kernels, each consisting of three Verilog files (top-level, data path, and sequencer) plus a directory containing netlists and simulation models for each hardware operator instance, and a C software output file containing the original C code augmented with HW/SW interfacing. The reconfiguration scheduling pass *does* create a data path load graph, but this graph is *not* used in any way to create FPGA bitstreams or to actually schedule reconfigurations.

Kasprzyk presents no runtime measurements of the generated hardware.

A simulation environment is available, but this is rather rudimentary: Only one hardware kernel can be simulated at a time, and quite a lot of manual intervention is necessary to run a simulation. For example, the I/O register numbers have to be manually found in the Verilog code and inserted into the top-level simulation file.

The implementation also lacks robustness; even simple input programs can cause it to crash.

The COMRADE 1.0 source code is to some extent (mainly in the SSA and the back-end passes) non-transparent and partly redundant, making it difficult to maintain.

Finally, the toolchain is restricted to integer computations, i.e., there is no support for floating point operators and data types.

# 4. Related Work

In this Chapter we will first categorize existing adaptive compilers and high-level synthesis frameworks for the C language and review five representative approaches (Section 4.1). Section 4.2 then discusses related work on early evaluation with cancel tokens, which is a central feature of COMRADE.

## 4.1. Existing Adaptive Compilers and HLS Frameworks

Today there are many frameworks which can create hardware descriptions from C or C-like representations. Comparisons are difficult to some extent, because commercial compilers understandably do not publish internal matters and the developers do not use consistent benchmarks. In the field of adaptive computing, another issue is that different compilers often use different target architectures that lack a commonly accepted test basis.

As our own compiler COMRADE focuses on the generation of dynamically scheduled hardware from unrestricted, non-extended ANSI C, we make a comparison to the existing approaches by concentrating on two central aspects: the kind of scheduling (static vs. dynamic) and the support of arbitrary pointers. We choose the latter because arbitrary pointer support is a feature which is typically lost when the supported C is restricted to some subset of the language. Table 4.1 accordingly categorizes 21 commercial and academic compilers which have emerged during the past 15 years. It is notable that most compilers use static scheduling. Dynamic scheduling has hitherto been employed by academic compilers, except for the Xilinx CHiMPS approach.

The next Sections highlight five representative compilers. Despite interesting approaches and features, none of the compilers known to us (beyond COMRADE) have ever examined early evaluation with cancel tokens (Section 4.2), which is a primary objective of this work.

*Static scheduling:*

| **Compiler** | **arbitrary pointers** | **Compiler** | **arbitrary pointers** |
|---|---|---|---|
| Bach-C [STKY99] | no | Handel-C [Bowe09] | no |
| C2H [LaPM06] | yes | Impulse-C [Impu11] | no |
| Catapult-C* [Ment09] | no | Mitrion-C* [Mitr11] | no |
| CHC [Alti08] | yes | NIMBLE [MacM01] | yes |
| Cyber [Waka99] | no | PRISC [RaSm94] | yes |
| Dime-C* [Stef08] | no | ROCCC [GuNB08] | no |
| FPGA C [Bass11] | no | SPARK [GDGN03] | no |
| GarpCC [CaHW00] | yes | xPilot [CFHJ06] | no |
| GAUT [GCHB05] | no | | |

*Dynamic scheduling:*

| **Compiler** | **arbitrary pointers** | **Compiler** | **arbitrary pointers** |
|---|---|---|---|
| CASH [Budi03] | yes | COMRADE [KoKa05] | yes |
| CHiMPS [PBDM08] | yes | Molen [Pana07] | no |

Table 4.1.: Existing C to hardware compilers (without claim of completeness). *The fact that Catapult-C, Dime-C, and Mitrion-C use static scheduling is very likely, but not 100% sure from the documentation and publications.

### 4.1.1. PRISC

One of the earlier approaches (1994) is PRISC (PRogrammable Instruction Set Computers), developed by researchers of Harvard University. It differs significantly from the other compilers, because rather than being used as a dedicated coprocessor, the generated accelerator hardware is programmed into the data path of the CPU. A hardware-programmable functional unit (PFU) such as this executes computations in parallel with the other CPU functional units (such as ALUs), as illustrated in Fig. 4.1. For simplicity, the logic in the PFU is limited to a latency of one CPU clock cycle. To use the PFU, the CPU instruction set is extended appropriately. The drawbacks of this approach are the limited memory access of the PFU (only possible using the existing CPU memory interface) and the limitation to combinatorial logic in the PFU. Simulations assuming a PFU in a 200 MHz CPU have shown relatively low speed-ups between 1.06 and 1.91 for SPEC'92 benchmarks.

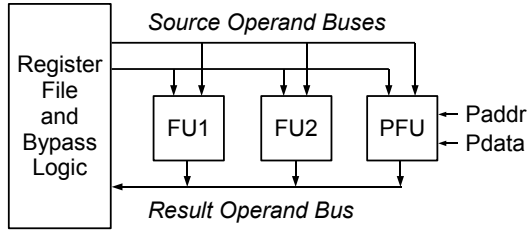


Figure 4.1.: The PRISC computation model. Source of image: [RaSm94], page 173, Fig. 1.

### 4.1.2. ROCCC

A prime example of statically scheduled hardware accelerators used as a co-processor is ROCCC (Riverside Optimizing Compiler For Configurable Computing) [BuNa08, GuNB08], developed at the University of California, Riverside. This is a hardware-only compiler, generating VHDL from designated C source code passages. It is built on the SUIF2 framework [ADHL00] and uses Machine-SUIF virtual machine [SmHo02] as low-level IR. Machine-SUIF is an assembler-like, data flow-based representation. ROCCC extends Machine-SUIF by foi instructions (feedback or initialization), which feed data paths in loops either with initial values (when the loop starts) or with feedback values from the previous loop iteration. Fig. 4.2(a)) shows a sample Machine-SUIF

code, representing a loop header in the upper section (Node:2) and parts of the loop body in the lower one (Node:3). The data path generated from this code is shown in Fig. 4.2(b). The fois are implemented as multiplexers controlled by loop controllers. Like many statically scheduled compilers, ROCCC separates computation processes and memory accesses from each other, i.e., the data path contains no elements accessing memory. Thus, the C source code is very restricted in relation to memory accesses. Pointers are not allowed at all, all array indices must be computed from the loop index plus a constant stride, arrays represent exclusive read-only or write-only accesses, and arrays may not have more than two dimensions. In the presence of these restrictions, ROCCC uses smart buffers which reduce the number of memory reads by re-using data from previous loop iterations. ROCCC creates efficient systolic array hardware accelerators. This results in very high speed-ups up to 3000x on a Virtex-4 @ 174 MHz when compared to a Xeon @ 2.8 GHz. On the other hand it imposes a lot of restrictions on the input C code, decreasing the language support significantly.

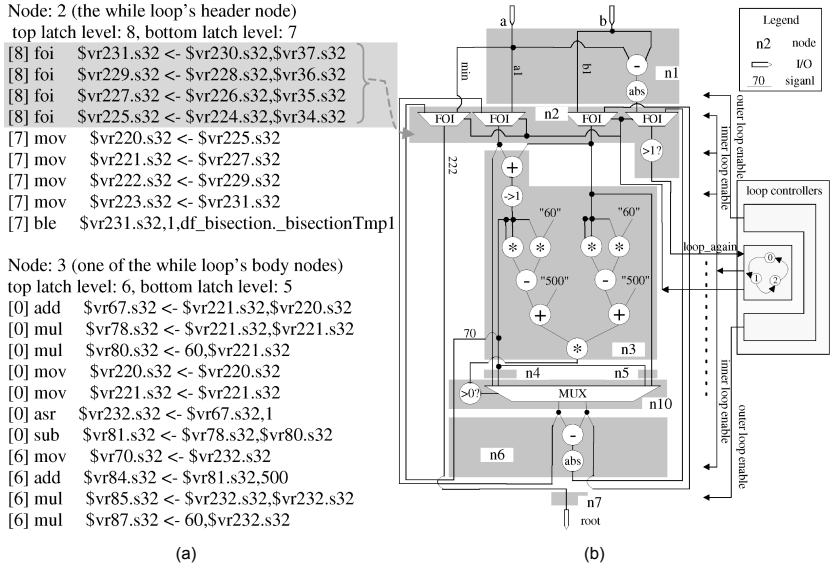


Figure 4.2.: (a) Machine-SUIF representation and (b) data path generated by ROCCC. Source of image: [GuNB08], page 6:12, Fig. 6.

### 4.1.3. SPARK

The SPARK compiler [GGDN04] developed at the University of California, Irvine, compiles C to RT-level VHDL code. Similar to ROCCC, SPARK uses static scheduling and does not allow pointers in the input C code at all. The main focus of SPARK is the evaluation of different optimization strategies such as (speculative) code motion and dynamic common subexpression elimination[1] [GDGN03]. Therefore, the published results focus on comparisons of different optimization passes and their combinations rather than measuring hardware accelerator speed-ups.

SPARK uses a hierarchical IR called **hierarchical task graph** (HTG). An HTG is a directed acyclic graph having three types of nodes: single nodes, compound nodes, and loop nodes. A single node contains a sequence of program statements without intermediary branches similar to the basic blocks of a CFG. Compound nodes contain sub-nodes and thus account for the hierarchical character of HTGs. Loop nodes are special compound nodes containing a loop head, loop body, and loop exit node. Each of these are again either a single or a compound node. The nodes are connected by directed control flow edges. Hence, the HTG essentially is a hierarchical CFG. Fig. 4.3 shows two example HTGs representing an if block (a) and a for loop (b).

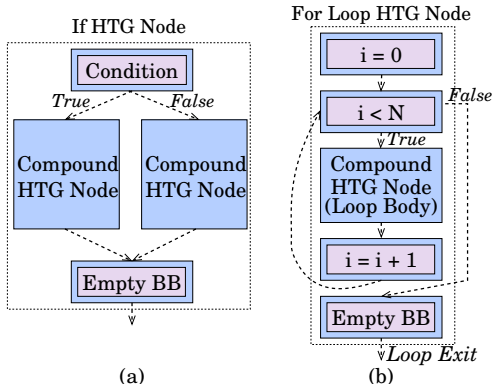


Figure 4.3.: Hierarchical task graph (HTG) representation of (a) an if block and (b) a for loop. Source of image: [GDGN03], Fig. 2.

[1] Common subexpression elimination carried out during the static scheduling phase.

### 4.1.4. CASH

As far as we know the first compiler generating dynamically scheduled hardware from C is CASH (Compiler for Application-Specific Hardware) [Budi03] developed in 2003 by Budiu (Carnegie Mellon University, Pittsburgh). Like ROCCC it is a pure hardware compiler, however instead of synchronous hardware descriptions for reconfigurable devices it creates asynchronous descriptions destined for ASIC synthesis. The concept of spatial computation described by Budiu originates from data flow computers. Operations are executed as soon as all of their inputs are available. CASH assumes a monolithic cached memory architecture, thus the support of arbitrary pointers in C is not a problem. The focus of Budiu's work is **Pegasus**, a CDFG-based intermediate representation. Fig. 4.4(B) shows an example Pegasus graph which represents the code of Fig. 4.4(A). Nodes contain operations for data transformation and token handling. Edges forward data, boolean values (used for predicated execution) and tokens. A Pegasus graph is built from a CFG and consists of connected subgraphs generated from different CFG regions (e.g., loops). Merge and eta nodes represent data inputs and outputs for each Pegasus subgraph.

Budiu's work contains concepts of speculative execution of data paths and even speculative memory accesses using a load-store queue (LSQ). However, many of these concepts have only been evaluated in high-level simulations (i.e., simulating Pegasus models) lacking actual HDL implementations. Low-level Verilog simulations use a memory system without LSQ and assume a perfect cache. Furthermore, the Verilog back-end does not allow function calls or pipelining of operators.

High-level simulations against a CPU exhibiting the same latencies as the generated hardware have shown speed-ups between 0.5x and 12x. In contrast, post-layout simulations of kernels mapped to an asynchronous ASIC using a [180nm/2V] standard cell library from STMicroelectronics against a 600 MHz CPU showed slowdowns of 0.78x up to 4.5x. According to Budiu the major reason for this slowdown is the memory access protocol, which does not allow the issue of a new access before the current access has completed.

### 4.1.5. CHiMPS

CHiMPS (Compiling High level language to Massively Pipelined System) [PBDM08], developed by Xilinx, is one of the latest compilers found in literature. As a C to hardware/software tool, it creates dynamically scheduled hardware, borrowing the spatial computation model introduced by CASH. The

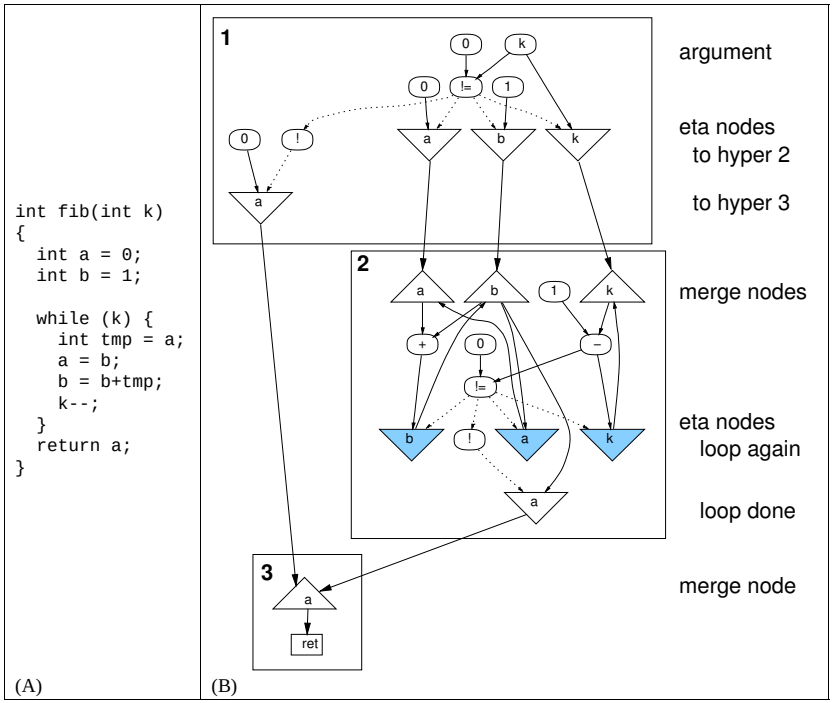


Figure 4.4.: (A) Sample C source code. (B) Pegasus representation (simplified). Source of image: [Budi03], page 48, Fig. 3.13.

fundamental CHiMPS innovation is the many-cache model. To increase the number of parallel cache accesses, the cache is split up into multiple caches that are synchronized through flushing. The mapping of memory access operations to caches uses a simple memory analysis based on the C99 `restrict` keyword. Compared to its first version, which did not carry out any optimizations during compilation, CHiMPS has offered an enhanced front-end based on LLVM [Latt02] since 2008. This offers better C language support and more high-level optimization passes [LeRK08].

Fig. 4.5 illustrates the main compilation steps. The input C code (a) is transformed into the CHiMPS Target Language (CTL) (b). CTL is an assembler-like IR describing data flow (shown graphically in (c)) as well as memory dependences between operations. From CTL representations CHiMPS then creates VHDL code.

The target platform most recently used for HW/SW applications generated by CHiMPS is the Xilinx Accelerated Computing Platform (ACP). The used ACP configuration connects a Xilinx Virtex-5 LX110T FPGA to an Intel quad-core Xeon CPU @ 2.66 GHz via the Intel Front-Side Bus (FSB). Cycle-accurate simulations of this platform have shown application-level speed-ups between 1x and 67x[2] against the ACP using the Xeon without FPGA support [PEBD09].

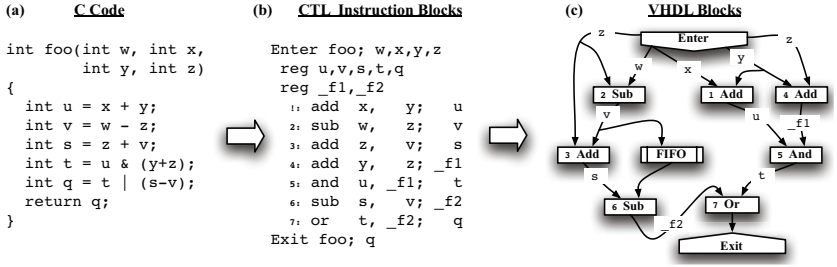


Figure 4.5.: (a) Sample C function; (b) CHiMPS Target Language (CTL) representation; (c) graphical DFG representation. Source of image: [PBDM08], Fig. 1.

[2]The assumed operation frequencies of the kernels running on the FPGA have not been published.

## 4.2. Early Evaluation and Cancel Tokens

In dynamic scheduling a hardware operation is executed when all of its inputs are available. Early evaluation (EE) of the operation can save execution time: The operation is executed and possibly outputs a result even though not all of its inputs are available yet. For example, if the selected data input of a multiplexer is available, the value is already forwarded to the data output, although the other inputs are not ready yet. However, to implement such a behavior, we have to handle the data that finally comes in through the pending inputs. More precisely, we have to discard those data items to prevent them from being used as input for consecutive computations. Thus, if an early evaluation takes place, a token representing a discard information must be attached to each pending input. We call these tokens **cancel tokens** (CT), while the tokens representing valid data are called **activate tokens** (AT). CTs are either **static**, i.e., they wait at the pending inputs until they collide with an incoming AT, or they are **dynamic**, i.e., they move in reverse data flow direction until they collide with an AT. If an AT and CT collide they erase each other.

The ideas of early evaluation and cancel tokens have been developed independently by two authors. In 2003 Brej [BrGa03] described EE for asynchronous hardware on the gate level. He used the term **anti-token** instead of cancel token. In 2005 Kasprzyk [Kasp05, KoKa05] described EE for synchronous hardware, which is created by the COMRADE compiler from ANSI C code. He used the term **up token** instead of activate token and **down token** instead of cancel token.

Of course, implementing EE is reasonable only if the inputs of an operation can actually arrive at different points in time. When static scheduling is used, data is processed during fixed time slots and inputs always enter an operation simultaneously. Hence, EE is only used in dynamically scheduled hardware. It can save runtime if data paths are imbalanced (e.g., different numbers of pipeline stages in alternative branches) or if operations can have a variable latency.

# 5. COMRADE 2.0 Execution Model

In this Chapter we will first illustrate the generic target architecture of COMRADE 2.0 (Section 5.1). Then we will present our hardware/software co-execution model employed in the architecture (Section 5.2) before describing in Section 5.3 the execution model of the hardware generated by COMRADE 2.0.

## 5.1. Generic Target Architecture

COMRADE 2.0 relies on platform-independent concepts for hardware and software generation and is thus not limited to a specific target platform. However, some basic properties of an adaptive computer target architecture can be derived directly from the COMRADE 2.0 design goals. The most important issue is the broad language support including hardware generation from C code containing pointers. Pointer support implies that the hardware cannot be constrained to use just simple variables and arrays for data I/O. Instead, it must have access to complex, pointer-based data structures, e.g., an array of pointers each referencing other pointer arrays. In such scenarios copying the relevant data to dedicated HW-accessible memories before the actual HW kernel computation (and copying the results back afterwards) is difficult and, if at all possible, often inefficient. A better alternative is to just pass a pointer to the referred data structure from SW to HW, analogous to the pass-by-reference principle. Recent work [LaKo09] has shown that direct exchange of pointers between software and hardware can be implemented at high performance even in full-scale operating system environments using virtual memory. The resulting requirement for the target architecture is a shared memory for SW and HW which holds all data that is accessible by both parties. One way to implement this (and this is the method we employ in our subsequent tests) is to use a subsection of the main memory as shared memory and grant the RCU access to that memory section.

Ideally both the CPU and the RCU use the same cache for accessing the shared memory (Fig. 5.1(a)). This allows the RCU to directly load the data processed by the CPU from the cache and vice versa. Unfortunately, as far as we know, there are no real test platforms which offer such a configuration. Instead, existing platforms use a separate cache for the RCU (Fig. 5.1(b)), typically configured onto the FPGA. Cache coherency can be ensured either by cache coherency protocols (MESI, MOESI) implemented in hardware or through software mechanisms. The caches are connected to the shared memory via a crossbar switch.

Furthermore, the CPU and the RCU are directly connected to each other by two channels (Fig. 5.1). The CPU r/w channel allows the CPU to directly address the RCU, e.g., to exchange live variables between SW and HW. The IRQ channel is used by the RCU to signal the end of a HW computation to the CPU or to signal an interruption of the HW computation, initiating a SW service.

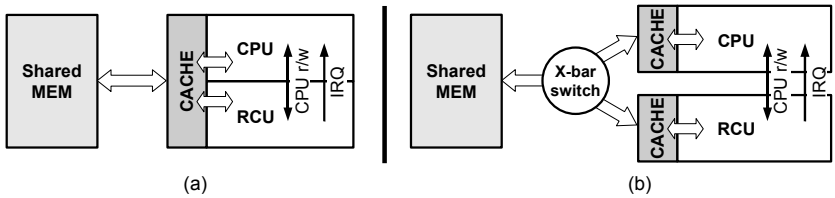


Figure 5.1.: Target architectures: (a) ideal, (b) real.

## 5.2. Hardware/Software Co-Execution Model

This Section uses an example to explain the HW/SW co-execution model, i.e., the way the CPU and the RCU behave and communicate to each other during the execution and the switching between SW and HW parts of an application. Fig. 5.2 shows the partitioned CFG of the sample program used throughout this Section. Although the CFG is only an intermediate representation, we will refer to CFG nodes or regions being *executed* on the adaptive computer; this actually means that the computations modeled by a CFG node or region are executed.

For this work we assume that regions are always executed in the original program order. As a consequence only one region is active at a time. Parallelization of regions would require sophisticated pointer analysis techniques (such

as [ShHo97]) to prove memory independence and is not in the scope of this work.

The example program in Fig. 5.2 comprises three SW regions (one of them being a SW subregion) and one HW region. Execution starts with node (0) on the CPU. After some initial assignments the HW region is entered. For such a SW/HW transition the SW performs four steps:

1. The CPU has to program the FPGA with the HW region to be executed if this has not been done before[1]. This step is not shown in the Figure.
2. The SW transfers the values of those variables to the HW, which are needed during the HW kernel execution (`SendVars(kernel_no)`).
3. If necessary it flushes the data which is stored in the shared memory from the CPU cache, if the adaptive computer has to manage cache coherency in software. This third step is also not shown in the Figure.
4. The SW starts the HW kernel (`StartHW(kernel_no)`) by writing to a dedicated hardware register of the target kernel. The CPU then waits until the RCU signals the end (or an interruption) of its computation via an IRQ.

Program control has now been transferred to the RCU, i.e., the hardware decides on its own when its computations are finished or when to interrupt them. This execution model is called **master mode**. However, note that although the RCU now controls the program flow and is able to directly access the memory, it never accesses the CPU except by sending an IRQ. Effectively, variable exchange between SW and HW is always initiated by SW.

After the HW has received the current values for `n`, `in1`, `in2`, and `out`, it enters a loop which is decomposed into the loop condition in node (2), the loop body (3, 4, 5, 6, 7, 8), the increment of `i` located in node (8), and a CFG edge $(8, 2)$ from the last body node back to the loop condition.

If `b == 0` in node (3), the HW interrupts the CPU and calls a SW service, i.e., SW subregion (5) is entered. To this end, the HW writes into a dedicated HW register called irqreg the number of the exit node (4) and then issues an IRQ (`StopHW(4)`)[2]. The SW interrupt handler, not shown in the Figure, reads

[1] Depending on the HW kernel sizes, multiple kernels can be arranged in one FPGA configuration to minimize the total number of reconfigurations. This is known as reconfiguration scheduling and has been explored by Kasprzyk in [KaVK05].

[2] In absence of a cache coherency mechanism, the RCU has to flush its cache before issueing the IRQ.

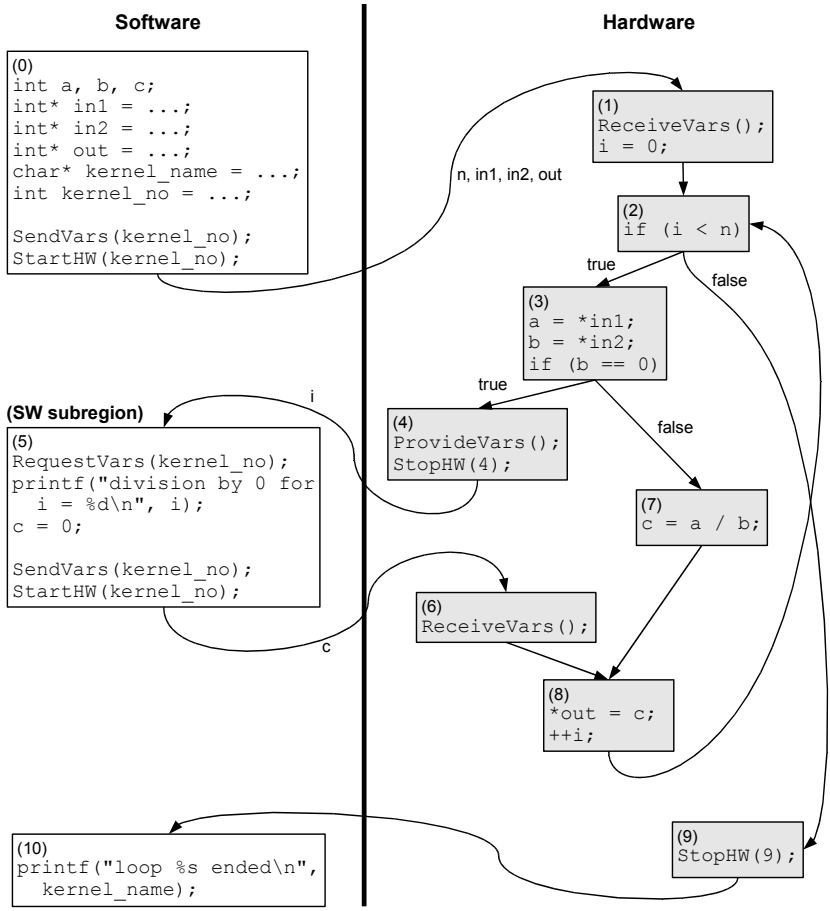


Figure 5.2.: COMRADE co-execution model with SW service. Numbers in brackets denote CFG node numbers.

value 4 from the irqreg and, using an internal map from IRQ reasons to SW basic blocks, therefore knows that SW execution has to continue with CFG node (5). The software reads the value of variable `i` from the HW, handles the exception, and continues HW execution. Note that the HW remembers that it has been stopped at CFG node (4) and so knows where to continue on the next `StartHW(kernel_no)` call.

Further interruptions may occur until the hardware loop finally ends and node (9) is reached. After the final HW/SW transition (9, 10), node (10) is executed in SW and the program finishes.

## 5.3. Hardware Execution Model

One of the main goals of COMRADE 2.0 is the implementation of a hardware generation methodology which supports a wide range of the ANSI C language, including arbitrarily nested conditions, loops, and pointers. As a side condition, the hardware generated should of course have a high runtime efficiency so that it actually makes sense to use it as a HW accelerator. For this, COMRADE 2.0 exploits several techniques:

- For better performance, even in the presence of variable latency operators (e.g., cached memory accesses) COMRADE 2.0 uses dynamic scheduling (Section 2.3).
- Through speculative predicated execution (Section 2.2) alternative branches (if/else, switch) and loop bodies are precomputed. This technique, originally applied for pure SW execution on CPUs, is especially amenable to data flow-based computation in hardware, because all branch targets can be precomputed simultaneously and without the risk of a pipeline stall. We refine speculative predicated execution by early evaluation (Section 4.2).
- Operations with a latency $> 1$ (even those with variable latency) are pipelined according to their specific initiation intervals (instead of using a loop-wide worst case).

After explaining the basic elements of the COMRADE hardware execution model (Section 5.3.1) we will illustrate that model with several representative examples, relating input C snippets to compiler-generated hardware models (Section 5.3.2).

### 5.3.1. Modelling Dynamic Scheduling with Early Evaluation

To be able to model mutual influences of data, control, and memory dependences and to support pipelined operators we have developed the *COMRADE Controller Micro Architecture (COCOMA)*. Before the next Chapter gives a formal definition, we will introduce the essential elements of COCOMA and explain the model using some illustrative examples.

A COCOMA instance can be pictured as a directed graph; the nodes correspond to C operations (e.g., arithmetic), while three distinct kinds of edges represent data, control, and memory dependences between them. Nodes can store and forward data items as well as two different kinds of tokens: activate tokens (AT) and cancel tokens (CT) (cf. Section 4.2). Edges can forward ATs and CTs and they are able to buffer one token. Memory edges, however, buffer or forward only ATs. Data edges forward data in addition to tokens. ATs moving along data edges always reference a (possibly speculative) piece of data, which is stored in a node and moves from node to node along with an AT. CTs move in reverse data flow direction. If AT and CT collide they cancel each other out. If all ATs referencing a piece of data are eliminated, that piece of data is deleted. Control edges can have different annotations which affect the token flow.

Fig. 5.4 shows graphical representations of the most important COCOMA elements. All operations storing or manipulating data (apart from multiplexers) are displayed as ellipses containing the operator name; an optional number below the name denotes the current output value. To start such an operation all incoming edges need to have an AT, with one exception: If there are multiple incoming control edges an AT from one of them suffices.

The same ellipse symbol with a dashed frame is used for token nodes; these only propagate tokens, not data. Different kinds of token nodes, identified by different names, perform different operations on tokens.

Data edges are solid; control and memory edges are dashed and have a different thickness. Control edges can have different annotations, influencing their semantics as explained in the examples below.

For multiplexers we use the traditional trapezoid symbol. COCOMA multiplexers support early evaluation, i.e., they can forward a value even if not all inputs have valid data yet, indicated by ATs. We distinguish three different kinds of multiplexers: muxes, loop muxes, and irq data muxes.

**Muxes** merge data paths after an if/else or switch branch. Each data input is

associated a one-hot encoded select value, printed below the incoming data edge inside the mux. The annotation of the incoming control edge defines how the select signal is computed from the result of the control edge source node. An example is shown in Fig. 5.3. Each value in the annotation ==(2,19,4) is compared to the result of node n. Only for the value 4 the comparison is true, giving the result vector 0,0,1, which is exactly the current select value. Thus, the rightmost input is selected.

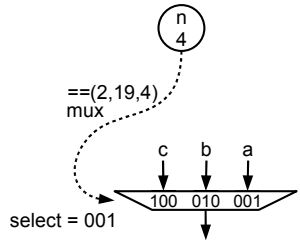


Figure 5.3.: Determination of the mux select signal.

**Loop muxes** (the word loop is printed inside the trapezoid) represent the data entry and exit points of loop bodies. A loop mux always has exactly two inputs: an init input i and a continue input c. The latter receives data from the loop body, the former from outside the loop. There is also an incoming control edge, but here only the continue input is controlled by the edge. There is no select signal; the loop mux reads the init input initially and after the loop ending and accepts the continue input while the loop is iterating.

**Irq data muxes** merge exit codes for IRQ signaling at HW/SW transitions. They neither have a select signal nor an incoming control edge. Here, simply the active input is propagated. COCOMA makes sure that only one input of an irq data mux can be active at a time.

ATs are printed as circles containing a +; CTs analogously contain a -.

The runtime behavior of the COCOMA elements regarding the token flow for the dynamic CT model is shown in Fig. 5.5. Note that the purpose of these rules will become more apparent in Section 5.3.2, where the greater context is described.

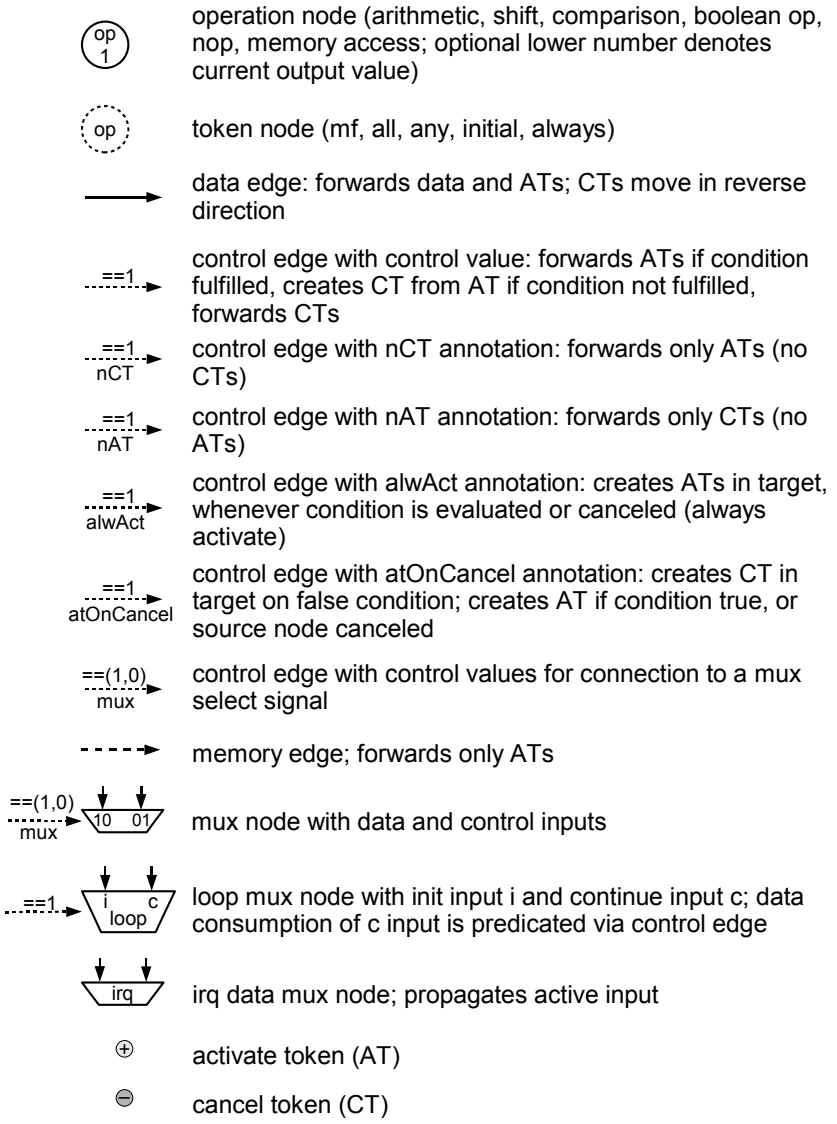


Figure 5.4.: Basic COCOMA elements.

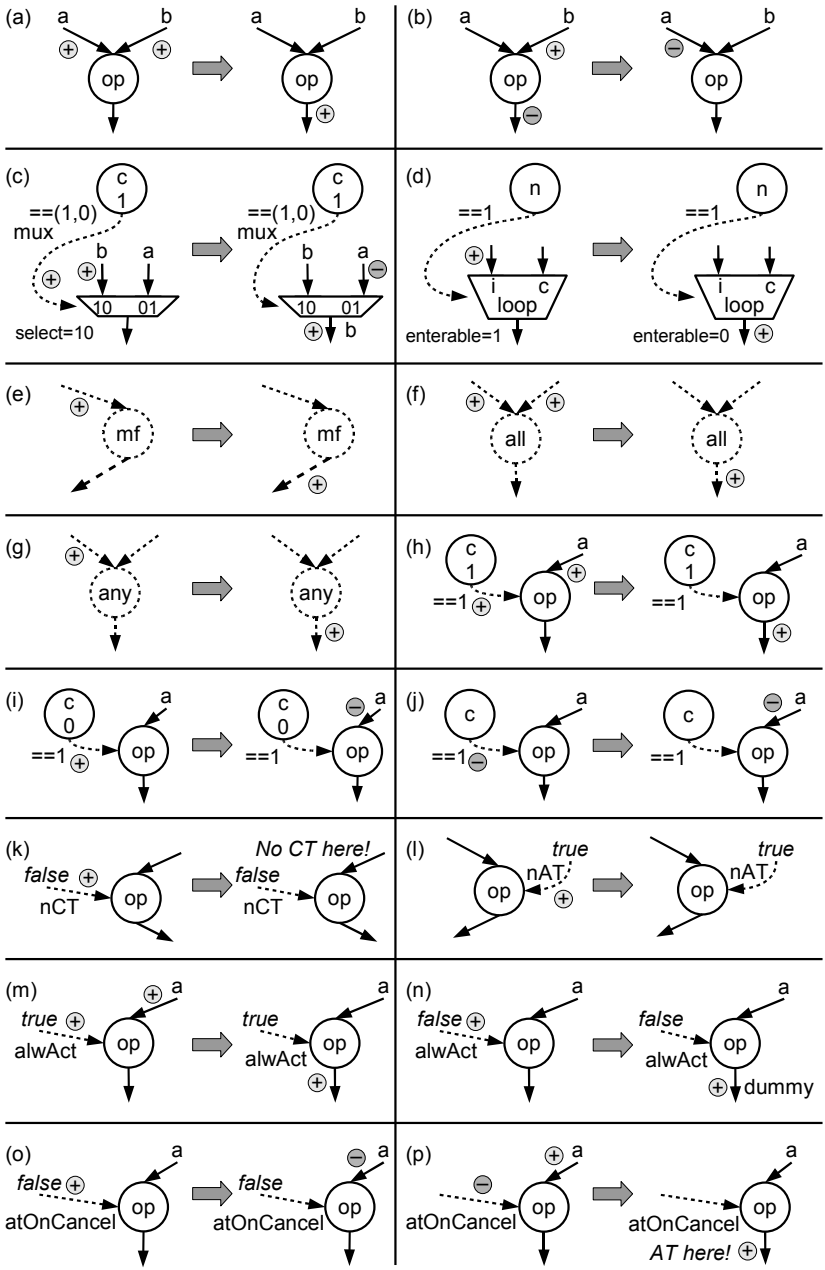


Figure 5.5.: COCOMA elements: token flow (dynamic CTs).

(a) represents the activation of an operation node: all inputs are active, op consumes the ATs and starts its computation. When it finishes it assigns an AT to each outgoing edge.

In Fig. (b) op is canceled and propagates the CTs to its inputs. At input b, the CT extinguishes an AT.

(c) shows an operator controlling a mux. The control edge annotation ==(1,0) establishes that if c is active and its current result is 1, input 10 (here b) is propagated through the mux and input 01 (a) is canceled.

Loop muxes (d) initially accept the init input without consuming a token from the controller node n, which controls only the continue input. After being initialized the loop mux sets its internal enterable state to 0, refusing additional initializations and accepting one value from the continue input per loop iteration. This corresponds to one AT coming in via the control edge per iteration. An incoming CT signals the loop mux the end of the loop, provoking the loop mux to reset its enterable state to 1. Section 5.3.2 (Loops) explains why this mechanism does not inhibit pipelining.

(e) shows the handover of an AT from a control edge to a memory edge. This is performed by a mf (memory forwarder) token node.

Note that for a given COCOMA node data dependences are fulfilled if *all* data predecessors deliver an AT[3], whereas control dependences are fulfilled if *one* of the control predecessors delivers an AT. This behavior seems inconsistent at first glance, but it makes sense when investigating actual COCOMA instances because this behavior is simply required in most situations. A more consistent notation would require additional COCOMA nodes and possibly lead to (slightly) more complex hardware. However, if the default behavior shown in Fig. (g) is not desired, control edges can be combined in an all node (Fig. (f)) requiring all control inputs to deliver an AT before an AT is sent via the outgoing control edge.

Fig. (h) explains another application of control edges: the predicated execution of a (non-mux) operator. If the control annotation matches the output of c and all incoming edges have an AT, op is executed. Fig. (i) shows the case in which the c output does *not* match the annotation. Here, op is canceled instead. Thus, the AT of the control edge (c, op) turns into a CT because the control condition is not fulfilled ($0 \neq 1$). If operator c is canceled already (regardless of its result value), that CT is forwarded to op as depicted in Fig. (j). This shows that CTs not only flow in reverse direction along data edges but also in *forward*

[3] Multiplexers are an exception; here, already one active predecessor can activate the multiplexer.

direction along control edges. Note that the controlled operator decides when to commence the operation, given that the data inputs have an AT. Operations without side effects can take place before a token has arrived at the input control edge; other operations (e.g., memory accesses) have to wait for an AT at the control edge.

Figs. (k) to (p) show the effects of further control edge annotations on the token flow. An nCT annotation (Fig. (k)) establishes that the control edge does not create CTs: The token on the control edge simply disappears. Analogously, nAT annotations (Fig. (l)) prohibit the propagation of ATs. An alwAct annotation preserves the standard control edge behavior if the control condition is true (Fig. (m)), but in the case of a false condition (Fig. (n)) or canceled condition, an AT is created instead of a CT. Note that this AT is bound to dummy data which cannot be used for any reasonable computation. However, such an AT can be used to eliminate a left-over CT[4]. Edges annotated with atOnCancel (Figs. (o), (p)) forward an AT to the target node if the source node is canceled. The difference to the alwAct annotation is, that in the case of a false condition (with AT in the source node) a CT is sent to the target node (instead, an alwAct annotation would send an AT to the target node in this case).

## 5.3.2. COCOMA Equivalents for Software Constructs

### Sequential Code without Intermediate Branches

Fig. 5.6(b) shows the COCOMA section created from the code snippet in Fig. 5.6(a). Each operator will execute its computation as soon as all inputs are available, i.e., all incoming data edges have an AT. At data path branches (fan-out), such as the branch after the + operator, an AT is assigned to each outgoing edge so that the successors / and − can start their computations independently. However, the two ATs created by the + are bound to a single piece of data: the result of the addition.

### If/Else and Switch Branches

The mux in Fig. 5.7 joins the data flow after an if/else or switch branch. At the same time it represents the end of a speculative data flow section. The mux does not consume a (speculative) data input before the correct input has been chosen by the control edge. The mis-speculated inputs are canceled; in the dynamic

[4] A statement on the use of this technique follows in Section 8.3

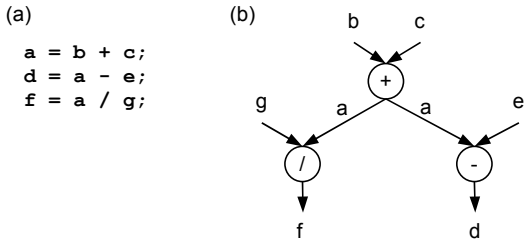


Figure 5.6.: Sequential operations: (a) sample code snippet, (b) generated COCOMA section.

CT model (cf. Section 4.2) they are assigned a CT which will move in reverse data flow direction until it extinguishes an AT. In the static CT model the CTs are stored inside the mux until the mis-speculated data actually arrives.

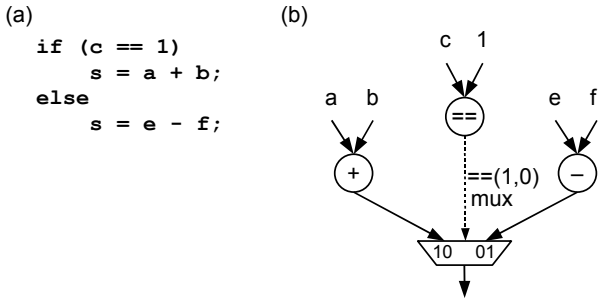


Figure 5.7.: If/else branch: (a) sample code snippet, (b) generated COCOMA section.

### Memory Accesses

Fig. 5.8 shows an example with two memory writes. Because these operations have side effects, they are not executed speculatively; therefore they are predicated with an incoming control edge.

Memory edges are used to enforce the execution of memory accesses in program order. An access may not start before the incoming memory edge (if any) has an AT, i.e., the previous access has completed. However, a direct memory edge between the two store nodes in Fig. (b) would raise a problem. If `c` was

false, the store *p would not be executed and thus not emit an AT to the store *q. *q would wait for *p to complete, resulting in a deadlock. As a solution we insert memory forwarder statements (<mf>) in the C code in the front-end of COMRADE 2.0, from which we generate memory forwarder (mf) nodes in COCOMA as shown in Fig. 5.8. Of course, mfs are only needed if one target of a branch actually contains a memory access. Similar to a mux joining data flows, an any node joins alternative memory dependences and forwards the AT to successive memory accesses. We annotate control edges to mfs with nCT because creating CTs would not make sense here; mf nodes do *not* have data predecessors or control successors that have to be canceled.

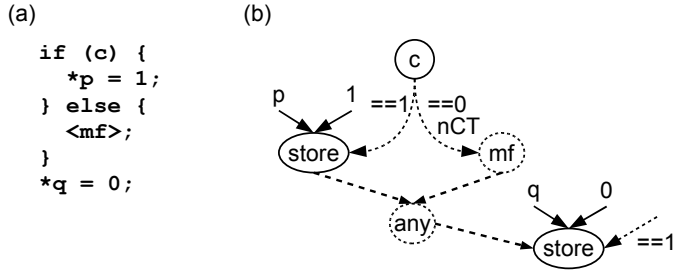


Figure 5.8.: Memory accesses: (a) sample code snippet, (b) generated CO-COMA section.

Fig. 5.9(a) shows a sample code containing a load and a store. Figs. (b) and (c) illustrate how tokens flow when a memory read receives a CT from a control edge. In (b) both control conditions are unfulfilled, because the output of c does not equal 1. This results in CTs generated for each data input of each of the memory accesses. The problem now is the CT moving from the store up to the load (assuming the dynamic CT model). There is no AT which could be canceled by the CT. Suppressing the generation of this CT is not an immediate solution, because there *is* an AT coming into the division operator from its second data input (75). Therefore, an AT is generated at the output of the load *although* the load is canceled. Note that this AT is bound to dummy data which will never produce a relevant computational result; its single purpose is to collide with the superfluous CT. We thus call such ATs **pseudo activate tokens (pAT)**. This technique also works for the static CT model.

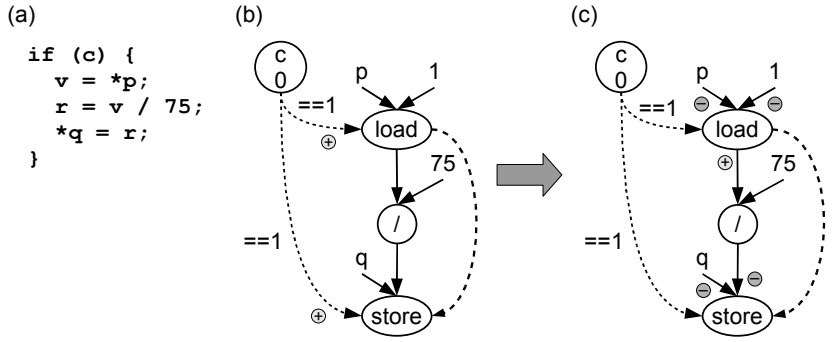


Figure 5.9.: Token flow for canceled loads: (a) sample code snippet, (b) COCOMA section with unfulfilled control conditions, (c) pseudo AT generation at load output.

### Nested Conditions

COCOMA has to guarantee that the store *p in Fig. 5.10 is only executed if both c1 and c2 are true. To guarantee this, COCOMA inserts a control edge from c1 to c2, reproducing the control hierarchy in the C code.

If c1 is false and creates a CT for c2, that token is forwarded along the outgoing control edges of c2 to prevent the store from being executed in the case c1 == 0 and c2 == 1. Thus, a CT is distributed in the whole conditional subtree.

This CT distribution raises a problem concerning the behavior of muxes. If c1 is false, a CT enters c2 and is then sent to the upper mux in Fig. 5.10(b), so that none of its data inputs is propagated to its output. But the lower mux anticipates an AT from each data input (including the upper mux) even in this case, resulting in an excess CT (or a missing AT) at the left input of the lower mux. As a solution, we require muxes to output a value in any case, even if all of its inputs are discarded; i.e., in that latter case, the mux emits a dummy value (which will be discarded by a subsequent mux). Alternatives to this approach are detailed in Section 8.3.

### Loops

Fig. 5.11 shows how loops are implemented with COCOMA. Here, looping is based on three general concepts. First, the **loop control node** (i.e., the node

(a)

```
s = 0;
if (c1) {
  if (c2) {
    s = a + b;
    *p = s;
  }
}
```

(b)

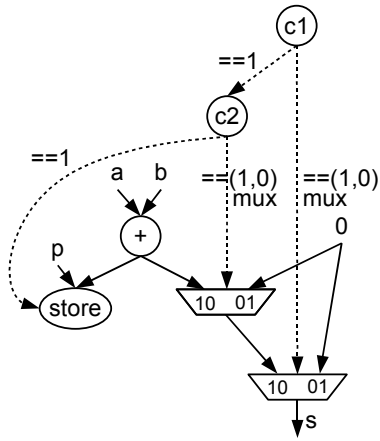


Figure 5.10.: Nested conditions: (a) sample code snippet, (b) generated COCOMA section.

which computes the loop condition) is evaluated $n + 1$ times for $n$ loop iterations, i.e., once for each iteration and once for realizing the loop end. This allows COCOMA to implement very complex loop conditions beyond simple constant-stride increments. From the loop control node a control edge leads to an any token node; another control edge leads back to the loop control node. This enables the reactivation of the loop control node after each loop iteration. If the any node was missing, the loop control node would need an AT from the control edge coming in from the left side in the Figure, which is not provided. However, if two control edges target a node, COCOMA requires only one of the control predecessors to have an AT in order to activate the target node (see details in Appendix A). As the any node is only used to forward ATs, we add an nCT annotation.

Second, each input of the **loop body data paths** (i.e., the set of operators originating from a loop body and connected via data edges) is assigned exactly one AT per token assignment of the loop control node. Token assignment includes that the loop control node is assigned an AT (evaluation of the loop condition) or a CT (forwarded from an outer condition or loop).

Third, speculation is extended to loop bodies. The loop body data paths start computing already before we know if the loop will conduct an(other) iteration.

Each variable which is assigned a value in the loop body and which is con-

(a)

```
s = 0;
n = ...;
k = ...;
for (i = 0; i < n; ++i) {
  s += 7;
}
r = s * k;
```

(b)

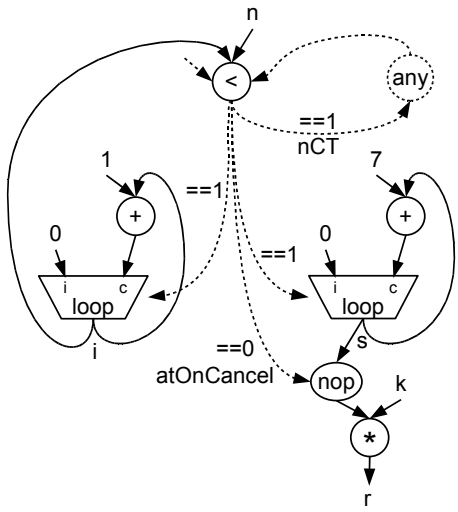


Figure 5.11.: Loops: (a) sample code snippet, (b) generated COCOMA section.

sumed during a successive loop iteration or by an expression after the loop is a **loop mux variable**. In Fig. 5.11(a) loop mux variables are i and s. Fig. (b) shows the COCOMA section generated for this code. For s, a loop mux has been generated. Its init input is connected to 0, while the continue input receives the updated value from an adder. A similar structure is generated for the loop index variable i; here, the value 1 (instead of 7) is added for each iteration.

According to its internal enterable state, the loop mux accepts data from its init input only when a loop starts, propagating the continue input afterwards. The initialization is independent of the loop control node, which controls only the continue input of the loop mux. As soon as a CT enters the loop mux via the control edge, the loop mux switches the enterable state back to 1 after all iterations have actually been processed.

Note that the data successors of a loop mux can be inside the loop body (e.g., the two adders) as well as outside (e.g., the nop node, which is a simple data buffer). While the adder consumes an AT from the loop mux during each loop iteration, only the final result of s may flow into the multiplier outside the loop. We handle this through inserting a control edge from the loop control node to each loop mux successor outside the loop body. The condition annotation is inverted

as shown in Fig. 5.11(b) (==0 instead of ==1), so that a CT is created for every valid loop iteration. Only when the loop control node is evaluated to 0, i.e., after the last loop iteration, is the loop mux output propagated to successors outside the loop. The atOnCancel annotation establishes that if the loop control node itself is canceled (e.g., because the loop is located inside an if branch which is not chosen), the initial value of the loop mux is forwarded to the data successors outside the loop instead being canceled. This is an important feature to justify the correctness arguments in Chapter 8. The nop node is inserted to make sure that the CT dedicated to the loop mux output does not cancel the k input of the multiplier. If the control edge led directly to the multiplier, a CT would flow into the k input for each loop iteration. This would produce excess CTs because k provides only *one* value intended to be multiplied with the *final* s value exiting the loop.

Fig. 5.12 shows how COMRADE guarantees that the variables which are read in a loop body, but are not assigned a value inside the loop, are reactivated, providing an AT with associated data for each loop iteration. The C code in part (a) contains the variables i and n, i being assigned a new value in each iteration due to ++i, but n not being assigned a new value. The COMRADE 2.0 front-end alters the code, producing the version shown in part (b). After the insertion of the n = n; statement (which never changes the program semantics), all variables used inside the loop body receive a new value in each iteration. Part (c) shows the generated COCOMA section with loop muxes for i and n. This approach, which has been borrowed from COMRADE 1.0, features quite a simple implementation (covered by a simple front-end pass), but introduces an area overhead: In a more sophisticated implementation, the loop mux for n could be optimized away. In that case, however, a mechanism for the proper reactivation of n would be needed. Note that simply keeping n active is not a solution because this would introduce excess ATs on the one hand and render updates of n (with a different value) impossible on the other hand.

Note that although the loop mux enterable state guarantees that the loop mux is not re-initialized before the entire loop has ended, pipelining loop bodies is *not* inhibited. On data paths along loop muxes, there is actually no pipelining; however, as long as loop-carried dependences are limited, it takes just a few cycles until a loop mux outputs the value for the next iteration. E.g., for the increment ++i of a loop index variable, two cycles are needed. Pipelineable loop data paths typically receive their input from a load node and write the output to a store node; both of which are part of the loop body. Such paths also need input from loop muxes; but these quickly deliver new data (e.g., every two cycles), *independently* of the length of the pipelined data path. Thus, the loop

mux mechanisms do not inhibit pipelining.

(a)

```
n = ...;
for (i = 0; i < n; ++i) {
  ;
}
```

(b)

```
n = ...;
for (i = 0; i < n; ++i) {
  n = n;
}
```

(c)

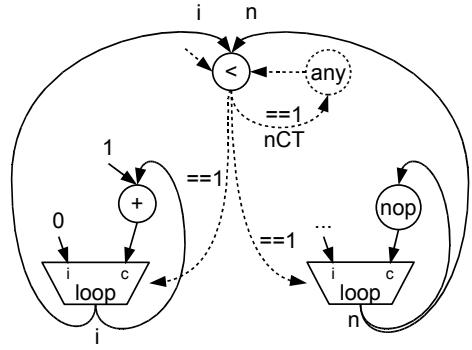


Figure 5.12.: Data reactivation in loops: (a) sample code snippet, (b) code altered by compiler front-end, (c) COCOMA section generated from (b).

### Nested Loops

Fig. 5.13 shows a nested loop example. Here, we have to guarantee that the outer loop does *not* introduce a CT (via CT forwarding along control edges) into the inner loop control node *before* the inner loop has finished, i.e., the inner loop control node has been evaluated $n + 1$ times for $n$ inner iterations. Otherwise a running inner loop would be incorrectly terminated early.

This is established by delaying the evaluation of the outer loop control node. We replace the original any node by a network of token nodes that outputs an AT as soon as all inner loops have ended. In Fig. 5.13(b) this reduces to a simple any node which is activated when the inner loop has ended ($(j < m) == 0$) or the inner loop has not been started at all ($(i == 4) == 0$). The control edges leading into this network are annotated with nCT, because this computation is only AT-related. If there are multiple inner loops, an all node is used in the token node network to guarantee that *all* inner loops have ended before the next outer loop iteration.

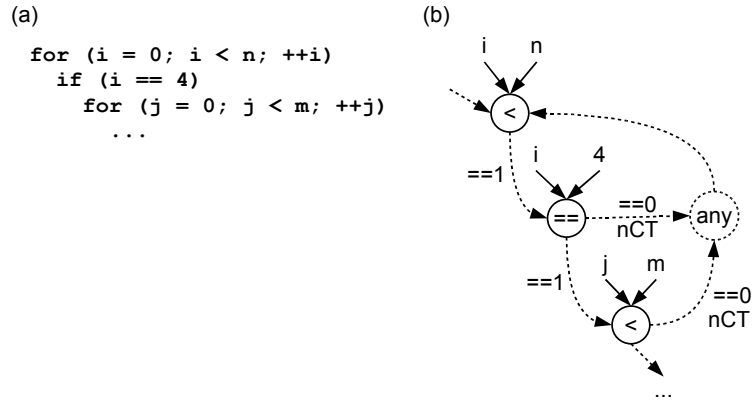


Figure 5.13.: Nested loops: (a) sample code snippet, (b) generated COCOMA section.

### Memory Accesses in Nested Loops

Fig. 5.14 shows dependent memory accesses on different loop levels. store3 is executed if $i < 8$, but not before the inner loop has finished. Here, the program order of the memory accesses cannot be established by a memory edge from store2 to store3, because only the execution of store2 in the *last* inner iteration is relevant. Instead, store3 is controlled by two control edges, one from the outer loop control node being responsible for creating CTs in store3, and one from the inner loop control node being responsible for AT delivery. We encode this with a nAT annotation of the edge that creates CTs (no ATs) and use nCT annotations for the edge delivering the ATs.

Fig. 5.14(b) also shows the general way used by COCOMA to obey memory dependences between loop iterations. The store2 accesses have to be carried out in program order so that store2 in the last iteration of the inner loop is executed after all previous store2 accesses. For this we insert a memory edge from store2 back to the loop control node (via an any node). We call the resulting structure of sequential memory accesses and control nodes a **memory chain**.

### Hardware/Software Interface

To receive variables transferred from SW to HW, a COCOMA instance has input register nodes (**inreg**). An inreg receives an AT as soon as a data word

(a)

```
for (i = 0; i < 8; ++i) {
  // load1
  s += *(p + i);

  for (j = k; j < 9; ++j) {
    // store2
    *q = s + j;
  }

  // store3
  r[i] = i;
}
```

(b)

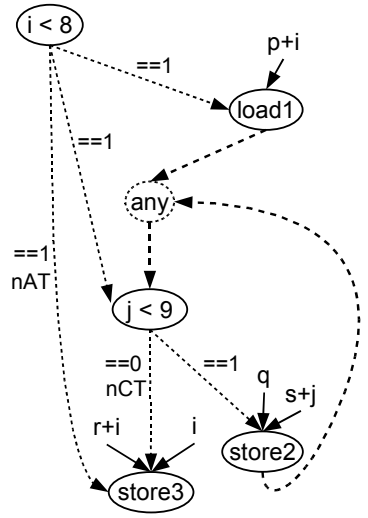


Figure 5.14.: Memory accesses on different loop levels: (a) sample code snippet, (b) generated COCOMA section.

has been transferred from the SW. For the other direction **outreg** nodes are used; from there the SW can read the currently stored value at any time. For signaling the end of the HW computation (or a break in HW computation in the case of a SW service call) an **irqreg** node is used. Receiving an AT, this node generates an IRQ for the CPU. The data word stored in the irqreg identifies the point in the program where the SW has to continue execution. This identifier can be read by the SW similar to outregs.

Fig. 5.15 shows an example containing a SW service which is executed if `b == 0`. If this condition is true in Fig. (b), the outreg waits for the `c` value, which it stores, and then activates the const node. The number 4 in this case is the identifier for the program section containing the branch target of `b == 0`, i.e., the section where the HW switches back to SW[5]. The identifier is forwarded through an irq data mux to the irqreg. All constant nodes identifying a HW-to-SW transition are connected to this special multiplexer, which simply forwards the currently active input. The irqreg now sends an IRQ to the SW. The SW has an interrupt handler (not shown in the Figure) which reads out the irqreg value and jumps to the corresponding line of code. From there, several

[5]COMRADE 2.0 uses CFG node numbers as identifiers, cf. Fig. 5.2.

outreg values may be read out, before the SW service executes (layered gray in Fig. 5.15(a)). After its completion several variable values may be transferred to inregs (`SendVars(kernel_no)`) and the SW switches back to HW (`StartHW(kernel_no)`). This last action activates an **initial** node. We assign such an initial node to each SW-to-HW transition and embed them in the memory dependence chain. This is necessary to ensure the program order execution of memory accesses even at the level of HW/SW regions. In the example, the store `*q = 9`, which is implemented in HW, may not execute before the store `*c = 0` in the SW service, which is guaranteed by the memory edge from initial to any.

(a)

```
if (b == 0) {
  ProvideVars();
  StopHW(4);

  RequestVars(kernel_no);
  printf("set %p = 0\n", c);
  *c = 0;
  SendVars(kernel_no);
  StartHW(kernel_no);

  ReceiveVars();
} else {
  *c = a / b;
}
*q = 9;
```

(b)

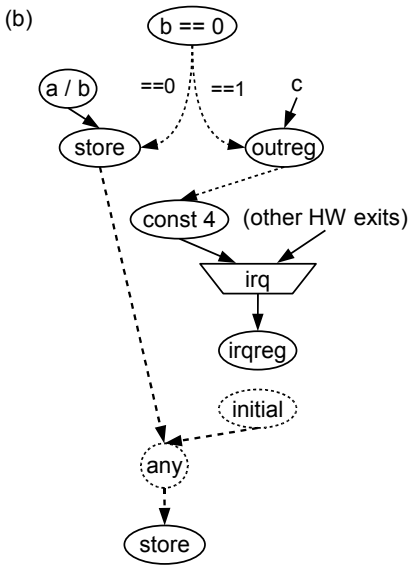


Figure 5.15.: HW/SW interface: (a) sample code snippet (SW service on gray layer), (b) generated COCOMA section.

# 6. COCOMA

The COMRADE Controller Micro-Architecture (COCOMA) is a token-based computation model and, at the same time, an intermediate representation (IR) for compilation from structured CFGs to hardware. From a COCOMA instance descriptions of dynamically scheduled hardware can be created in a straightforward way. COCOMA models variable latency operators, operator-internal pipelining, and early evaluation with cancel tokens (both dynamic and static CTs are supported). This Chapter gives a formal COCOMA definition, which serves as a baseline for the actual implementation in COMRADE. Because the complete model description is quite complex, we have moved some details to Appendix A.

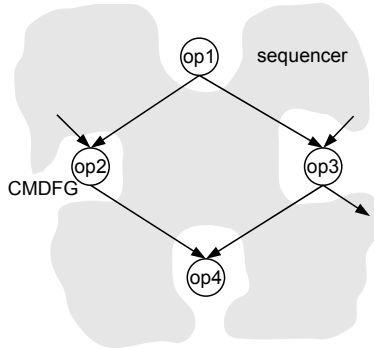


Figure 6.1.: COCOMA structure consisting of CMDFG and sequencer.

A COCOMA instance consists of a control memory data flow graph (CMDFG) and a sequencer as illustrated in Fig. 6.1. The CMDFG models parameterized operations and their dependences in a directed graph, while the sequencer defines token flow directives and manages the inter-operator handshaking. The sequencer can be automatically generated from the CMDFG, thus the CMDFG alone is an IR already - in the compile flow, it is the input of the compiler pass which generates the COCOMA. Despite this redundant character of the

sequencer, its generation is an important step towards hardware implementation.

In the sections that follow we will define the CMDFG (Section 6.1) and the sequencer (Section 6.2). As the CMDFG is a complete IR, its definition contains far more details than traditional graph-based descriptions such as DFG or CDFG. For example, nodes have a type defining their function and parameters specifying buffer sizes, bitwidths, and other options. I/O ports depend on the node type. Data edges are connected to I/O ports instead of nodes, which is essential for non-commutative operations. Section 6.3 then combines CMDFG and sequencer into a COCOMA instance.

## 6.1. Formal CMDFG Definition

First, we adopt two practical notations from [Golz78] for binary relations, as we will often make use of these.

**Definition 6.1.1.** Given a binary relation $E \subseteq A \times B$ and some set $C$, we define the **set of E-successors of C**, $\tilde{E}(C)$:

$$\tilde{E}(C) := \{b \in B : \exists a \in C : ((a, b) \in E)\}.$$

Dually, the **set of E-predecessors of C**, $\underset{\sim}{E}(C)$ is defined to be

$$\underset{\sim}{E}(C) := \{a \in A : \exists b \in C : ((a, b) \in E)\}.$$

For example, in a directed graph $G = (N, E)$, with $E \subseteq N \times N$, $\tilde{E}(\{c\})$ is the set of successor nodes of node $c$, i.e., the set of all nodes $b$ such that there is an edge $(c, b) \in E$. $\underset{\sim}{E}(\{c\})$ is the set of predecessor nodes of node $c$, i.e., the set of all nodes $a$ such that there is an edge $(a, c) \in E$.

We now define a *CMDFG frame* which imposes the CMDFG graph structure.

**Definition 6.1.2.** A **CMDFG frame** $CF = (N, E_{dat}, E_{con}, E_{mem})$ defines a finite directed graph $(N, E)$ with edge set $E = E_{dat} \dot{\cup} E_{con} \dot{\cup} E_{mem}$ partitioned into three disjoint subsets: **data edges** $E_{dat} \subseteq N \times N$, **control edges** $E_{con} \subseteq N \times N$ and **memory edges** $E_{mem} \subseteq N \times N$.

Each node has either *only* outgoing control edges or *no* outgoing control edges: $\forall (n, m) \in E_{con} : \forall k \in \tilde{E}(n) : (n, k) \in E_{con}$.

Note that this definition of a CMDFG frame does not allow more than one edge in the same direction between two nodes, i.e., a CMDFG frame is a graph, not

a multi graph. Let us shortly make sure that this restriction is acceptable for CMDFGs which represent sections of C code, by examining all the prohibited cases of edges between two nodes $n, m$.

If there is a control edge $(n, m)$ the CMDFG uses $n$ as dedicated control node. There will be no necessity for any data or memory flow from $n$ to some other node, because C statements can only be control dependent on an if/else or switch condition, not data or memory dependent. Thus, control/data or control/memory combinations of edges are not necessary. The only remaining combination of different edges is data/memory. Such combinations are not necessary either because a memory dependence would be redundant to an existing data dependence. Thus, we only have to examine multiple data edges, multiple control edges and multiple memory edges between two nodes $n$ and $m$.

Multiple data edges can only occur when the same variable is used as primary and secondary input of a binary operator, e.g., $a + a$. Nearly all of these cases can be replaced by other expressions (Table 6.1), thereby bypassing the problem. However, there are no direct substitutions for $a * a$ and $a << a$. For these cases (the latter being rarely used), an intermediate nop (no operation) node as shown in Fig. 6.2(c) can be inserted (without affecting the hardware latency when configuring the nop operation as combinatorial logic), avoiding the need for multiple data edges (Fig. 6.2(b)).

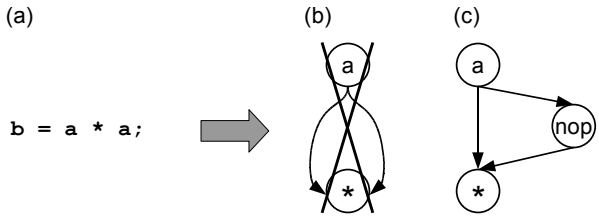


Figure 6.2.: Intermediate node insertion to prevent multiple data edges with the same orientation between two nodes.

Due to the structuredness of the C source code multiple control edges can only be created from fall-through switch statements as shown in Fig. 6.3(a). Instead of implementing multiple control edges (Fig. 6.3(b)), the solution is to aggregate the control values (0, 1) into a single control edge (Fig. 6.3(c)).

Furthermore, there is no need for multiple memory edges, because these would encode exactly the dependence modeled by a single memory edge.

On a CMDFG frame, we define some auxiliary functions:

| Problematic expression | Substitute |
|---|---|
| $a + a$ | $a << 1$ |
| $a - a$ | 0 |
| $a * a$ | — |
| $a / a$ | 1 |
| $a \% a$ | 0 |
| $a << a$ | — |
| $a >> a$ | 0 |
| $a < a$ | 0 |
| $a > a$ | 0 |
| $a <= a$ | 1 |
| $a >= a$ | 1 |
| $a == a$ | 1 |
| $a != a$ | 0 |
| $a \& a$ | $a$ |
| $a \&\& a$ | $a != 0$ |
| $a \mid a$ | $a$ |
| $a \mid\mid a$ | $a != 0$ |
| $a \hat{} a$ | 0 |

Table 6.1.: Replacement of C expressions which, when directly translated into CMDFG syntax, would lead to multiple data edges with identical orientation between two nodes.

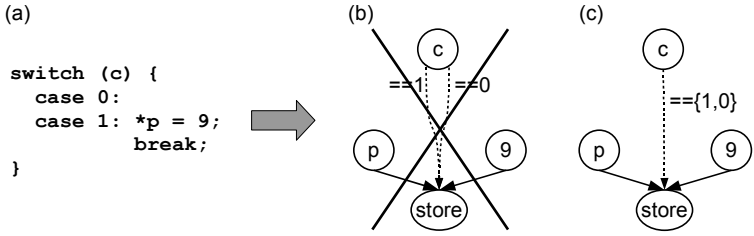


Figure 6.3.: Agglomeration of control values in a single control edge.

**Definition 6.1.3.** Given a CMDFG frame $CF$ with node set $N$ and edge set $E \subseteq N \times N$. The **outgoing edges function** $E_{out} : N \to \mathcal{P}(E)$ assigns each node $n \in N$ the set of its outgoing edges $E_{out}(n) := \{n\} \times \tilde{E}(n)$. The **outgoing control edges function** $E_{out,con} : N \to \mathcal{P}(E_{con})$ analogously assigns each node $n \in N$ the set of its outgoing control edges $E_{out,con}(n) := \{n\} \times \tilde{E}_{con}(n)$.

The **source projection function** $proj_1 : E \to N$ projects an edge to its source node $proj_1((n, m)) := n$.

Analogously, the **target projection function** $proj_2 : E \to N$ projects an edge to its target node $proj_2((n, m)) = m$.

We now define node types that can occur in a CMDFG, as well as input and output ports necessary for data flow.

**Definition 6.1.4.** A **node library**
$NL = (T, D_{in}, D_{out}, d_{in}, d_{out}, C_{in}, C_{out}, Param, param)$ consists of

- a set of **node types** $T$ reflecting all integer operations that can occur in a C source code, as well as auxiliary and token flow node types: $T :=$ {add, sub, mul, div, mod, negation (!), complement (~), shiftLeft, shiftRight, lessThan, greaterThan, lessThanOrEqual, greaterThanOrEqual, equal, notEqual, bitwiseAnd, logicalAnd, bitwiseOr, logicalOr, bitwiseXor, bitSel, mux, loopMux, irqDataMux, nop, inreg, outreg, irqreg, load, store, initial, always, all, any, mf},
- two non-empty, finite, disjoint sets $D_{in}$ and $D_{out}$ of **data input ports** and **data output ports**: $D_{in} := \{A, B\}$, $D_{out} := \{R\}$ (more than two data inputs, needed for multiplexers, are also supported – see below),
- a function $d_{in} : T \longrightarrow \mathcal{P}(D_{in})$ assigning a set of data input ports to each node type,
- a function $d_{out} : T \longrightarrow \mathcal{P}(D_{out})$ assigning a set of data output ports to each node type, obeying the condition $|d_{out}(t)| \leq 1$ for all $t \in T$, i.e., each node type has at most one data output port,
- two sets of input and output token flow ports: $C_{in} :=$ {Start, StartCtrl, StartCtrlPseudo, Sel, StartSel, ResultReadyAck, Cancel, CancelStateAck, CancelStateCtrlAck} and $C_{out} :=$ {StartAck, StartCtrlAck, StartSelAck, ResultReady, CancelAck, CancelState, CancelStateCtrl, TokensEmpty},
- a set of **node type parameters** $Param :=$ {WA, WB, WR, Sign, Depth, QDepth, TQDepth, StaticCT, StartCtrlIn, PseudoAT, NoCT, NIn, IOAddr, IOAWidth, PortNum,

| **Node Type** $t$ | $d_{in}(t)$ | $d_{out}(t)$ |
|---|---|---|
| add, sub, shiftLeft, shiftRight, lessThan, greaterThan, lessThanOrEqual, greaterThanOrEqual, equal, notEqual, bitwiseAnd, logicalAnd, bitwiseOr, logicalOr, bitwiseXor mul, div, mod, load | $\{A, B\}$ | $\{R\}$ |
| negation (!), complement (~), mux, loopMux, irqDataMux | $\{A\}$ | $\{R\}$ |
| store | $\{A, B\}$ | $\emptyset$ |
| bitSel, nop | $\{A\}$ | $\{R\}$ |
| inreg, const | $\emptyset$ | $\{R\}$ |
| outreg, irqreg | $\{A\}$ | $\emptyset$ |
| initial, always, all, any, mf | $\emptyset$ | $\emptyset$ |

Table 6.2.: Input and output data ports assigned to node types.

AddrWidth, DataWidth, Width, AreaNSpeed}, which define bitwidths, signedness, buffer sizes, the token flow model, and other properties (see Chapter 7 for details), and

- a **parameter assignment** $param : T \longrightarrow \mathcal{P}(Param)$, assigning a set of parameters to each node type.

$Mux$ := *{mux, loopMux, irqDataMux}* is the set of **multiplexer data types**.

We use $d_{in}$ and $d_{out}$ to assign input and output ports to the node types, so that there is a correspondence to C operators. For example, a *div* type has two distinct inputs (the dividend $A$ and the divisor $B$) and one output (the quotient $R$), corresponding to the C operator /. The distinction of the inputs is necessary because in general C operators are *not* commutative. Table 6.2 gives the exact definitions for $d_{in}$ and $d_{out}$. Note that inregs have no data input port in the CMDFG context. However, in the hardware implementation inregs are connected to a data bus, such that the CPU can feed data into the inreg, producing the new AT. Similarly, outregs and irqregs don't model data output ports. Multiplexer node types support an arbitrary number of inputs although they use only one data input port. All data inputs will be concatenated to a single bus, which is then connected to port $A$, having a greater bitwidth accordingly (cf. Fig. 6.5).

The following listing describes the semantics of the token flow signals, most of which implement a simple handshaking protocol. This handshaking is based on synchronous logic, i.e., in the hardware implementation each operator has

a Clock input which is not explicitly defined in the CMDFG. Further customary inputs are Reset and CE (clock enable) detailed in Secion 7.

- Start, StartAck: If Start is true, there are valid data items at the data input ports. If the node consumes the data, it sets StartAck to true. For multiplexer nodes, Start and StartAck are multi-bit signals, each bit respectively indicating or acknowledging a data item at exactly one data input.

- StartCtrl, StartCtrlAck: StartCtrl indicates an AT coming in along a control edge, acknowledged by StartCtrlAck.

- StartCtrlPseudo: If this signal is true while StartCtrl and StartCtrlAck are set, the node receives a pseudo activate token (pAT). Unlike ATs, pATs create a dummy value at the node data output without consuming a data input value. This can be used for the elimination of excess CTs.

- Sel: For mux nodes this is the (multi-bit) select signal. loopMux and irqDataMux nodes do not have an explicit select input.

- StartSel, StartSelAck: For mux nodes these signals are used to notify and acknowledge a valid value on the select input Sel.

- ResultReady, ResultReadyAck: If ResultReady is true, the node has finished the current operation. The value currently assigned to the data output port *R* (if present) represents the result of the computation. ResultReady and R are held until ResultReadyAck is true.

- Cancel, CancelAck: A true Cancel signals the node that the next result is superfluous. If CancelAck is true, the node will discard the next computed result or (if no computation is running and the dynamic CT model is used) forward the cancel information by setting CancelState and CancelStateCtrl to true.

- CancelState, CancelStateAck: A true CancelState forwards a CT to the data predecessor nodes, acknowledged by CancelStateAck.

- CancelStateCtrl, CancelStateCtrlAck: A true CancelStateCtrl forwards a CT to the control successor nodes, acknowledged by CancelStateCtrlAck.

- TokensEmpty: If true, the node does not currently contain any ATs or CTs. This information is needed to ensure that a loop has executed all its memory accesses before ending.

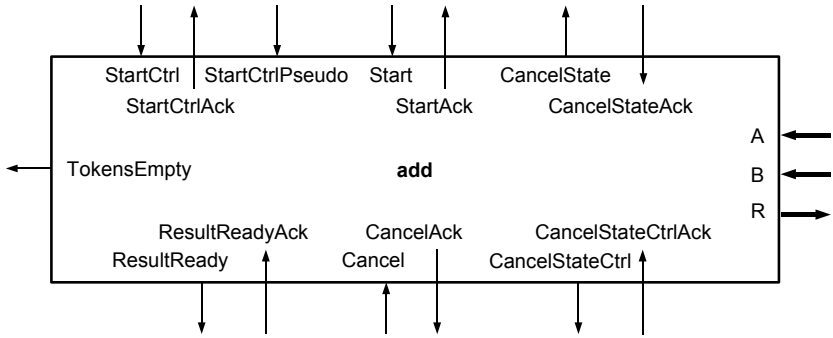


Figure 6.4.: Data and token flow ports for the node type add.

Fig. 6.4 shows the data and token flow ports relevant for the add node type.

Next, we assign node types to the nodes of a CMDFG frame, set parameters, and connect predecessor nodes to input ports.

**Definition 6.1.5.** Given a CMDFG frame and a node library, a **data flow** $DatF = (t, ps, pin, pout, bits)$ consists of:

- a **type function** $t : N \to T$, assigning each node a node type,
- a **parameter setting functional** $ps_n : param(t(n)) \to \mathbb{N}_0$, which sets the parameters of each node $n \in N$,
- an **input port functional** *pin*, which defines for each $n \in N$ a surjective input port function $pin_n : \underline{E}_{dat}(\{n\}) \to d_{in}(t(n))$, i.e., each data predecessor node of $n$ is assigned an input port of $t(n)$,
- an **output port functional** *pout*, which defines for each $n \in N$ a surjective output port function $pout_n : \tilde{E}_{dat}(\{n\}) \to d_{out}(t(n))$, i.e., each data successor node is assigned a data output port of $t(n)$, and
- a **bitwidth configuration** $bits : E_{dat} \to \mathbb{N}$, assigning each data edge a positive integer representing its bitwidth.

Given a CMDFG frame, a node library, and a node type $t_1$, the control edge set $E_{con}$ can be partitioned into two subsets $E_{con,t_1}$, $E_{con} \setminus E_{con,t_1}$, with $E_{con,t_1}$ containing all control edges $e \in E_{con}$ pointing to a node of type $t_1$.

A control flow assigns several attributes to control edges:

**Definition 6.1.6.** Given a CMDFG frame and a node library, a **control flow** $ConF = (ann, cw, muxPred, con, nctMux)$ consists of:

- an **annotation** $ann : E_{con} \to \{1, nAT, nCT, alwAct, atOnCancel, mux\}$, which can assign control edges an annotation *nAT* (no activate token), *nCT* (no cancel token), *alwAct* (always activate), *atOnCancel* (AT on CT), or *mux* (mux control), 1 representing no annotation,

- a **control width** $cw : E_{con} \to \mathbb{N}$, assigning mux control edges the number of the mux data predecessors, while the remaining control edges are assigned a 1, i.e.
$$cw((n,m)) = \begin{cases} \left|\underline{E}_{dat}(m)\right|, & \text{if } t(m) = mux \\ 1, & \text{otherwise,} \end{cases}$$

- a **mux predecessor assignment functional** $muxPred_m : \underline{E}_{dat}(m) \to \{0, \ldots, \left|\underline{E}_{dat}(m)\right| - 1\}$, for each mux node $m$ assigning each data predecessor a distinct number, which is needed below to associate predecessors and control functions,

- a **control configuration** $con : E_{con} \to \bigcup_{k\in\mathbb{N}} (F_{bool})^k$, which assigns each control edge $e \in E_{con}$ a vector of **control functions** with boolean domain, $\forall f \in F_{bool} : f : \mathbb{Z} \to \{true, false\}$, where $con(e) \in (F_{bool})^{cw(e)}$. If the target of $e$ is not a mux, $cw$ equals 1, so that the vector comprises only one element. The purpose of the control function is to tell for which outputs $z \in \mathbb{Z}$ of the control edge source node the successor node is activated or canceled, respectively. We use the set of integers here, because the branching of a switch statement can be based on any positive or negative integer value, in contrast to simple if/else scenarios, for which the set $\{0, 1\}$ would suffice. In the mux case the control function vector contains one control function per mux data input. Then, each control function determines for a given control node data output $z$, if the next value received from the associated mux data predecessor will be forwarded ($f(z) = true$) or discarded ($f(z) = false$). To associate control functions with mux data predecessors, *muxPred* is used; for mux $m$, control function $f_{i_j}$ in the vector $(f_{i_0}, f_{i_1}, \ldots)$ is associated to the data predecessor $p$ of $m$ such that $muxPred_m(p) = j$.

- A **mux control nCT annotation** $nctMux : E_{con,mux} \to \bigcup_{k\in\mathbb{N}} \{true, false\}^k$, with $nctMux(e) \in \{true, false\}^{cw(e)}$, can assign an nCT property to each mux input using the same association as the control configuration.

We can now define the CMDFG.

**Definition 6.1.7.** A Control Memory Data Flow Graph (CMDFG) $C = (CF, NL, DatF, ConF)$ consists of a CMDFG frame $CF$, a node library $NL$, a data flow $DatF$, and a control flow $ConF$.[1]

Given a CMDFG, a node configuration assigns values to the input or output ports of each node at a given abstract point $t$ in time[2]. Thus, this represents *dynamic* behavior in contrast to the static model described so far. Node input configurations are needed to define the current input values needed to carry out a computation in the nodes; node output configurations represent the computational results.

**Definition 6.1.8.** Given a CMDFG, a **node input configuration at time** $t$, $NI_t : N \times (D_{in} \cup C_{in}) \to \mathbb{Z}$, assigns each pair of a node and an input port an integral value. Token flow ports are always assigned either 0 or 1, with two exceptions: the Sel port transmitting the select signal for mux nodes, and the Start and CancelStateAck ports of mux and loopMux nodes. In these cases Start and CancelStateAck have a dedicated bit for each data input as shown in Fig. 6.5. However, such multi-bit signals (with each bit either 0 or 1) are represented by non-negative integers in the node input configuration.

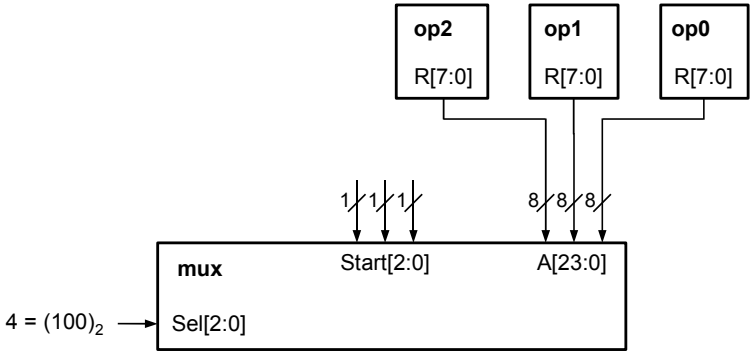


Figure 6.5.: Connection to multi-bit input ports; mux forwards next value coming in from op2, because associated Sel bit is 1.

By partitioning the domain, we break down a node input configuration at time

[1] There is no need for a dedicated memory flow component, because the memory edges (already being defined in the CMDFG frame CF) have no further annotations.

[2] Practically, time is discretized into clock cycles.

$t$ into a **node input data configuration at time** $t$, $NID_t : N \times D_{in} \to \mathbb{Z}$, and a **node input control configuration at time** $t$, $NIC_t : N \times C_{in} \to \mathbb{Z}$.

Analogously, a **node output configuration at time** $t$ $NO_t : N \times (D_{out} \cup C_{out}) \to \mathbb{Z}$ assigns each pair of a node and an output port an integral value. It is broken down into **node output data configuration at time** $t$, $NOD_t : N \times D_{out} \to \mathbb{Z}$, and **node output control configuration at time** $t$, $NOC_t : N \times C_{out} \to \mathbb{Z}$.

Having assigned values to the output ports, we can determine if the control functions which are assigned to control edges return a true or a false value. The following notation simplifies such return value queries.

**Definition 6.1.9.** Given a CMDFG, a node output configuration at time $t$, $NO_t$, and a node $n \in N$, we define the **evaluation functional** $eval_n : E_{out,con}(n) \to \bigcup_{k\in\mathbb{N}} \{true, false\}^k$, $eval_n((n,m)) := con((n,m))(NO_t(n,R))$.

Thus, for a control edge $e_{con} = (n,m) \in E_{con}$, assuming that node $m \in N$ is not a mux, $eval_n(e_{con})$ is true if the control function of $e_{con}$, i.e., $con(e_{con}) = f$, is true for the value assigned to the output port $R$ of $n$ according to the current node output configuration $NO_t$. Note that in general, $eval_n(e_{con}) \in \{true, false\}^{cw(e_{con})}$, because $con((n,m))$ is a *vector* of control functions, each one reading the same value assigned to the output port $R$ of $n$. Fig. 6.6(a) shows an example: Here, the current value assigned to output port $R$ is 0, leading to a true evaluation, because that value matches the control edge annotation. Formally, $e_{con} = (a,b)$, and $con(e_{con}) = f$, with $f : \mathbb{Z} \to \{true, false\}$, so that

$$f(R) = \begin{cases} true, & \text{if } R = 0 \\ false, & \text{otherwise.} \end{cases}$$

Because $R = 0$, $f(R) = \{true\}$, and therefore $eval_a(e_{con}) = true$. The example in Fig. 6.6(b) shows the (more complicated) mux case. As the mux has 3 data predecessors, $cw((c,d)) = 3$. For $e_{con} = (c,d)$, we have $con(e_{con}) = (f_2, f_1, f_0) \in F^3_{bool}$, with

$$f_2(R) = \begin{cases} true, & \text{if } R = 2 \\ false, & \text{otherwise,} \end{cases}$$

$$f_1(R) = \begin{cases} true, & \text{if } R = 19 \\ false, & \text{otherwise, } and \end{cases}$$

$$f_0(R) = \begin{cases} true, & \text{if } R = 4 \\ false, & \text{otherwise.} \end{cases}$$

As $R = 19$ in the example, $eval_c(e_{con}) = (false, true, false)$[3].

To simplify the notation, we may write $eval(n, m)$ instead of $eval_n((n, m))$.

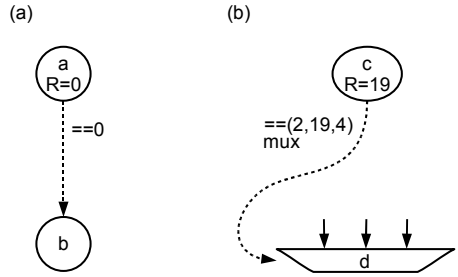


Figure 6.6.: (a) Non-mux control edge with control width 1; (b) mux control edge with control width 3.

Each CMDFG node is able to carry out a computation, i.e., use the values available at the input ports to compute resulting values which are then assigned to the output ports. We regard this **node operation** as a black box here, describing the internal computation structure of a node in Section 7. However, all CMDFG nodes together formally compute a node output configuration $NO_{t_2}$ from a node input configuration $NI_{t_1}$ as depicted in Fig. 6.7.

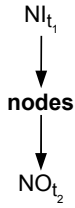


Figure 6.7.: Operation of CMDFG nodes.

## 6.2. Formal Sequencer Definition

Whilst data flow connections between input and output data ports (A, B, R) are already fully covered by the CMDFG, the sequencer defines logic which con-

[3] In practice, this will set the Sel input of the mux to 010 (binary).

nects the handshaking ports of adjacent nodes. The exact definition of this logic is quite complex for several reasons. First, data, control, and memory dependences influence each other. Second, the control edge annotations affect the token flow. Third, the token flow model is parametrized to support either dynamic or static CTs. Due to their complexity, refer Appendix A for sequencer details. In this Section we will concentrate on just the basic facts and definitions.

The logic defined by the sequencer depends on the direct neighborhood of a node. Fig. 6.8 shows the neighborhood of an arbitrary node $y \in N$. The predecessor nodes are contained in the node set $X$, which is partitioned into $X_{dat}$, $X_{con}$, and $X_{mem}$, containing data, control, and memory predecessor nodes respectively. According to the annotation $X_{con}$ can be further broken down into $X_{con,1}$, $X_{con,nCT}$, $X_{con,nAT}$, $X_{con,alwAct}$, $X_{con,atOnCancel}$, and $X_{con,mux}$. The same scheme is applied to name the successor nodes $Z$. Elements of such a node set are printed in lower case, e.g., $x \in X$. To clarify the associated node set, we sometimes use subscripts in node identifiers, such as $x_{dat} \in X_{dat}$.

To simplify the sequencer notation further, we write $InputPortName_t(y)$ instead of $NI_t(y, InputPortName)$ for referencing the value assigned to the input port *InputPortName* of node $y$, given a node input configuration $NI_t$. The analogous notation is applied for output ports. For example, given a node output configuration, $ResultReady_t(x)$ is the value assigned to the *ResultReady* output port of node $x$ at time $t$.

As a basic feature of COCOMA, successor nodes may consume incoming tokens independently of each other as illustrated in Fig. 6.9. $y$ provides a valid result in Fig. (a), which is consumed by $z_2$, but not yet by $z_1$ due to a missing token from $n$. Fig. (b) shows how we prevent $z_2$ from being started a second time with the same input token; $ReadyConsumed_{t_2}(y, z_2)$ is set to 1. Thus, while we block $ResultReady_{t_2}(y)$ from activating $z_2$, we let the signal pass to $z_1$, which still needs to be activated.

Considering current token positions in Fig. 6.9(a), $y$ places an AT on both data edges, $(y, z_1)$ and $(y, z_2)$, by setting $ResultReady_{t_2}(y)$ to 1. In Fig. (b) the AT on edge $(y, z_1)$ still remains, while the AT on edge $(y, z_2)$ has moved into $z_2$. Hence, an AT is present at an edge $(y, z)$ at time $t$, if $ResultReady_t(y) = 1$ and $ReadyConsumed_t(y, z) = 0$.

Analog arguments hold for CTs; here, $CancelConsumed_t$ prevents the CT from canceling a data predecessor more than once. We now formally define these two functions.

**Definition 6.2.1.** Given a CMDFG, a **sequencer configuration at time** $t$,

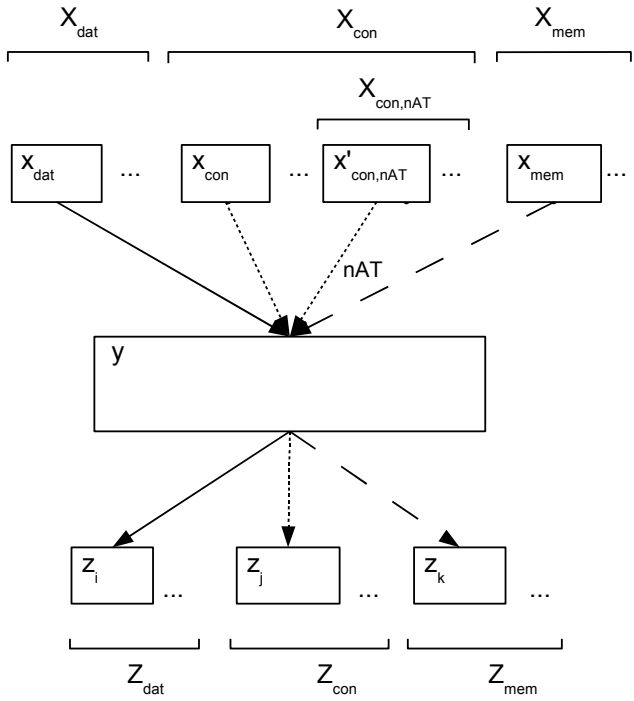


Figure 6.8.: Predecessors $X$ and successors $Z$ of a node $y \in N$.

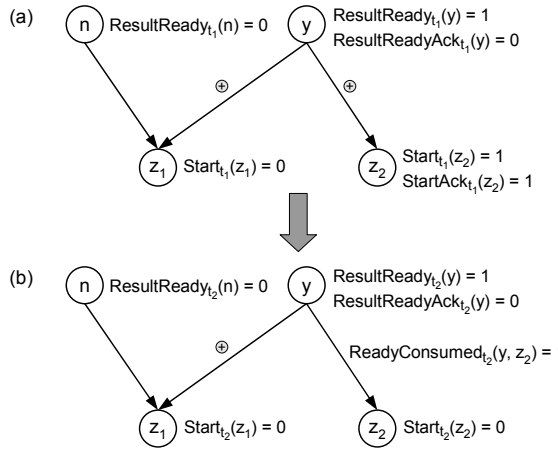


Figure 6.9.: AT flow example: (a) both outgoing edges of $y$ have an AT; (b) AT of edge $(y, z_2)$ has been consumed by $z_2$.

$SeqC_t$, consists of a **ReadyConsumed function** and a **CancelConsumed function**:

- $ReadyConsumed_t : E \rightarrow \{0,\ 1\}$,
- $CancelConsumed_t : E \rightarrow \{0,\ 1\}$.

The main duty of the sequencer is to compute a node input control configuration $NIC_{t_2}$ from a given node output control configuration $NOC_{t_1}$. It also needs to update its own sequencer configuration $SeqC_{t_1}$ to $SeqC_{t_2}$. This duty cycle is depicted in Fig. 6.10.

**Definition 6.2.2.** Given a CMDFG, a **sequencer** *Seq* reads a node output control configuration $NOC_{t_1}$ and a sequencer configuration $SeqC_{t_1}$, and outputs a node input control configuration $NIC_{t_2}$ and a new sequencer configuration $SeqC_{t_2}$. A detailed specification of how $NOC_{t_2}$ and $SeqC_{t_2}$ are computed is given in Appendix A. Furthermore, a sequencer is parametrized with a **scheduling type** *SchedType* which is either *dynCT* for dynamic cancel tokens or *statCT* for static cancel tokens.

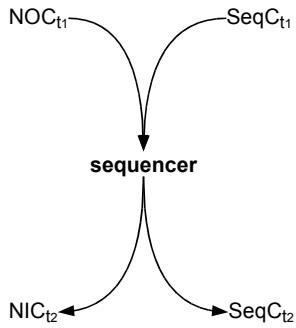


Figure 6.10.: The sequencer duty cycle.

## 6.3. Formal COCOMA Definition

To be able to model the behavior of combinatorial logic (e.g., for operation chaining), we will from now on use a more detailed notion of time. Instead of pure points $t$ in time, we write $(t, i)$, where $t$ can be understood as clock cycle number and $i$ references the delta cycle, which does not consume time in the model. Thus, $NOC_{0,1}$ and $NOC_{0,2}$ can be different, although they are assigned to the same clock cycle.

**Definition 6.3.1.** A **Comrade Controller Micro-Architecture (COCOMA)** instance $(C, NO_{0,0}, S, SeqC_0)$ consists of

- a CMDFG $C$,
- an initial node output configuration $NO_{0,0}$,
- a sequencer $Seq$, and
- an initial sequencer configuration $SeqC_0$.

To get a quick overview the COCOMA operation is shown in Fig. 6.11 in a simplified version without timing subscripts. The CMDFG nodes generate *NOD* and *NOC* from *NID* and *NIC*. The sequencer uses *NOC* and *SeqC* to compute *NIC* and update *SeqC*. Node data outputs *NOD* are directed to node data inputs *NID* according to the CMDFG data edges.

For the implementation of COCOMA in synchronous hardware, a central aspect is the mapping to actual clock edges. COCOMA primarily performs two

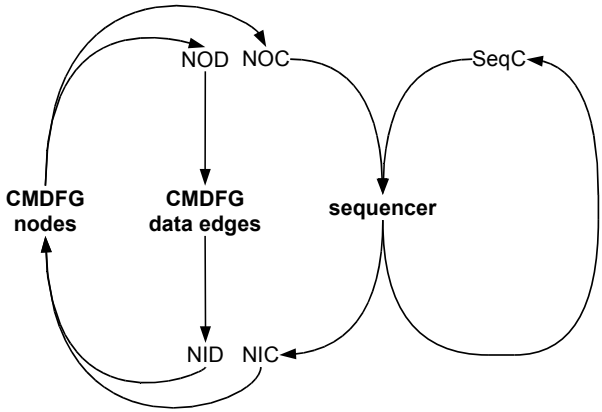


Figure 6.11.: COCOMA operation overview.

actions during one cycle. First, node outputs flow along data edges and through the sequencer logic to produce node inputs which are consumed by nodes to produce new node outputs. Second, the sequencer configuration is updated. However, mapping all of these operations to one clock cycle would restrict COCOMA too much, because at least one cycle would be needed to compute the node outputs from the node inputs. To support even combinatorial (unregistered) operations (operation chaining) COCOMA contains a delta cycle phase as shown in the upper part of Fig. 6.12. Here, new node outputs are computed until the configuration is stable. Whether a node works combinatorially depends on the node parameter DEPTH. DEPTH = 0 means a combinatorial computation (unregistered), DEPTH > 0 results in a non-combinatorial, buffered (registered) computation (cf. Section 7). After that, the lower part of Fig. 6.12 is entered increasing the time value ($m := n + 1$) and resetting the delta cycle number. Note that sequencer configurations *SeqC* do not need delta cycle updates, therefore only one index for clock edge mapping is present.

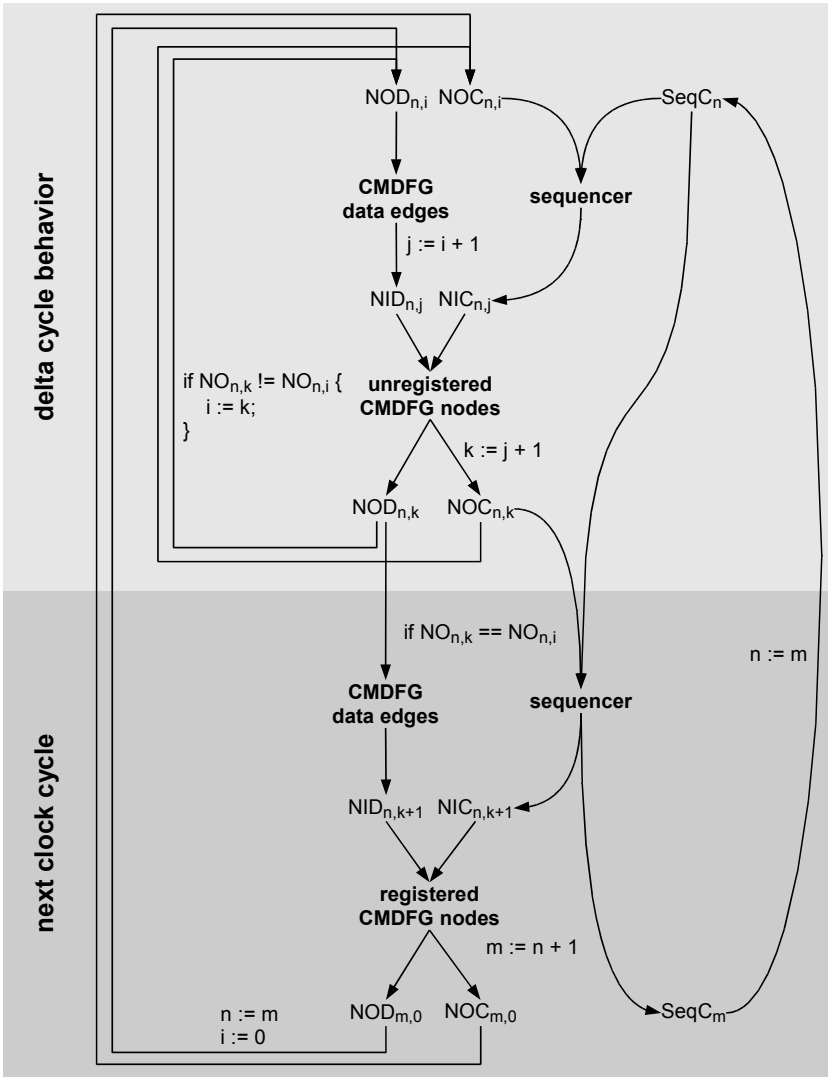


Figure 6.12.: Detailed COCOMA operation.

# 7. Hardware Operator Library

The COCOMA nodes motivated in Chapter 5 and defined in Chapter 6 require a hardware operator library from which they can be instantiated for simulation and actual synthesis purposes. COMRADE 2.0 sets out the following requirements for a library such as this:

- It must support all C language operators as well as the auxiliary and token flow operators defined in Def. 6.1.4.
- It is platform independent, i.e., it can be used for a variety of target FPGAs.
- Each operator is synthesizable, so that current FPGA synthesis, mapping and placement tools can automatically generate appropriate FPGA configurations.
- It is compatible to the COCOMA token flow, i.e., operators must be able to consume, process, and propagate ATs and CTs.
- It is parametrizable, so that operator features can be adjusted to the compiler needs. Parameters include bitwidths, buffer sizes, token flow (dynamic or static CTs), predication, and synthesis directives such as optimization strategies (e.g., area vs. time).
- It provides meta data for each operator, such as area consumption and achievable frequency, depending on the parameter values and on the target FPGA.

COMRADE 1.0 has used the GLACE module generator library [NeKo01] for the mapping of C operators to hardware. GLACE creates compact gate-level designs, furnished with placement directives for specific target platforms. This low-level approach is no longer required today. Logic synthesis tools have improved and at the same time the resources available on an FPGA have increased, so that the designing on the higher register transfer level (RTL) has become a universal standard. In particular the overhead of porting the GLACE

generator routines to new FPGA fabrics on the one hand and the additional features required by COCOMA on the other hand have prompted us to replace GLACE with a newer, higher-level and more powerful RTL operator library. This new library called **Modlib** has been developed fulfilling the above requirements in cooperation with the Embedded Systems and Applications Group (ESA), headed by Professor Andreas Koch at Technische Universität Darmstadt and Benjamin Thielmann (Abteilung E.I.S., Technische Universität Braunschweig), who has implemented the major part of the functionality.

Modlib is applicable beyond COMRADE. The library can also be utilized by compilers which create statically scheduled hardware. An example is a domain specific compiler for Geometric Algebra [HMSH10], which has recently been developed by the ESA Group at TU Darmstadt.

Section 7.1 describes the Modlib module parameters and highlights some of the library features. More Modlib details can be found in [GäTK10] and [ThGä10].

Modlib fulfills all of the above requirements except for the last requirement – it does *not* provide meta data. The GLACE generators used by COMRADE 1.0 contain an algorithm for each operator class (such as addition, multiplication etc.) which analytically computes the meta data. This approach is not feasible for Modlib due to the higher level of design. First, the actual gate level implementation of an operator is left to the synthesis tool, so that Modlib itself has no knowledge of the area required and frequency achieved on the target platform. Second, an analytical computation is not desirable because this would inhibit the immediate support of new target FPGAs; the analysis algorithms of each operator class would have to be updated for every new technology. Instead, meta data for Modlib operators are determined empirically. This is accomplished by the **Meta Data Fetcher (MDF)** framework, which has been developed in context of this work. Details about MDF are given in Section 7.2.

## 7.1. Modlib Parameters

Modlib consists of a number of Verilog files, with each file containing a module definition of one operator. This Section gives an overview of the module parameters and their meaning, highlighting some central features which are useful for the application of Modlib in COMRADE 2.0.

### WA, WB, WR

These parameters define the bitwidths of the data inputs A and B and the data output R.

### Sign

Sign = 0 instantiates an unsigned operator, Sign = 1 creates a signed operator.

### Depth, QDepth, TQDepth

These parameters define buffer sizes.

Depth stands for an optional additional latency, which can be useful for three purposes:

1. Implementation of a data output register. Most Modlib operators are combinatorial by default; to prevent long combinatorial paths in a sequence of several operators, COMRADE 2.0 adds an output register (i.e., Depth = 1) for each data manipulating operator. If two or more consecutive operators fit in one clock cycle, Depth can be locally adjusted to 0 (operation chaining).

2. Retiming: by setting Depth to a value greater than 1, the synthesis tool is given the possibility to span complex operations (such as multiplication or division) over several pipeline stages.

3. Pipeline balancing: A Depth value greater than 1 can be used to align the operator latency with the latency of a parallel data path.

QDepth defines the size of an optional, transparent data output queue. Transparent means that instead of inserting a fixed latency, the buffered data is forwarded immediately. If the queue is empty, data passes the buffer combinatorially; otherwise an incoming data word is stored until it exits the buffer. This can be used to alleviate back pressure. By having an output queue, an operator can compute further results, even though the current operator data output has not yet been acknowledged by the successive units. For dynamic scheduling contexts, this implements a dynamic kind of pipeline balancing. Thus, in dynamic scheduling, sections of fixed latencies can be balanced using both Depth or QDepth. Fig. 7.4 shows that (at least for the synthesis tool and the

target FPGA used here) for buffer sizes greater than one, transparent output queues (QDepth) occupy fewer area resources than fixed latency shift registers (Depth). Thus, we will generally use output queues for pipeline balancing issues in COMRADE 2.0.

TQDepth (used only when the operation is predicated) sets the size of a transparent buffer for tokens coming in via a control edge.

### StaticCT

StaticCT = 1 configures the operator for static CT behavior, i.e., CTs are not propagated to predecessor operators; StaticCT = 0 sets the dynamic CT behavior.

### StartCtrlIn

If StartCtrlIn = 1, the operator is predicated. It then executes its computation speculatively, but does not forward the result until an AT has entered the operator along an incoming control edge. Exceptions are the memory access operations memread and memwrite which do not access the memory before the AT has arrived[1]. StartCtrlIn = 0 means non-predicated operation, i.e., no control edge leads to the operator, and a result is output as soon as it is ready.

### PseudoAT

If PseudoAT = 1, the operator creates a pseudo AT (pAT) from an incoming CT, cf. Section 5.3.1, Fig. 5.5(n). PseudoAT = 0 conversely disables this feature.

### NIn

For multiplexers, NIn sets the number of data inputs. An exception is the loopMux operator, which always has two inputs.

[1] For memread and memwrite, we diverge from the speculative operator concept because of current restrictions of the implementation. As soon as our memory back-end supports speculative accesses, memread and memwrite may shift back in line and also implement the speculative execution semantics. Memwrite may also be updated to use the general mechanisms shown here.

### NoCT

For multiplexers configured for dynamic CTs, this parameter denotes the inputs to which no CTs may be propagated in the cancel case. COMRADE 2.0 needs this to avoid excess CTs, cf. Section 8.

### IOAddr, IOAWidth

These are parameters used only by the operators inreg, outreg, and irqreg, which implement I/O registers, i.e., registers which are accessible from an outlying authority, such as a CPU. IOAddr is the address associated to the register; IOAWidth is the bitwidth of the address bus.

### PortNum, AddrWidth, DataWidth, Width

These are parameters dedicated to the memory access operators memread and memwrite. PortNum defines the port number for memory systems with multiple parallel ports such as the MARC system [LaKo00] used by COMRADE 2.0. AddrWidth and DataWidth give the address and data bus widths; Width is used as a byte enable, supporting 8, 16, or 32 bits wide accesses.

### AreaNSpeed

If AreaNSpeed = 1, the operator is optimized for area; otherwise it is optimized for speed (i.e., for a high throughput with a low initiation interval). This does *not* change the parameters for the synthesis tool (such as the target frequency) – instead a preselection of the operator implementation is made in Modlib. This is currently only used for the division operator, which has by far the biggest spectrum of different implementations amoung the integer operators (cf. the comparisons in Section 7.2.1).

## 7.2. Meta Data Fetcher

Fig. 7.1 shows the MDF tool flow. When COMRADE 2.0 sends a query, MDF determines the desired meta data empirically. Query parameters include the operator name (e.g., add), the Modlib parameters described in Section 7.1, the target technology (e.g., Xilinx Virtex-5, xc5vfx70t, speed grade -1), the target

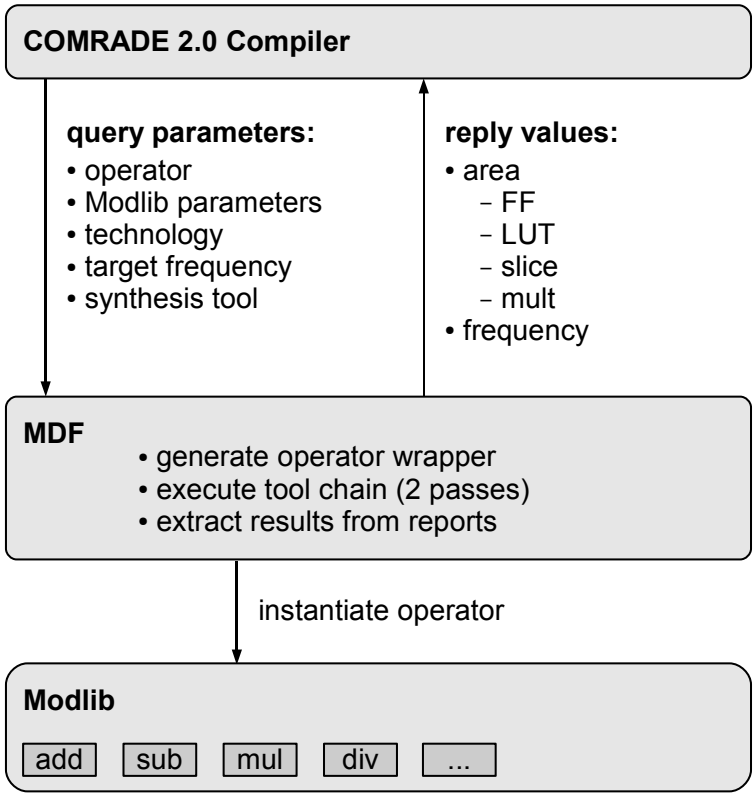


Figure 7.1.: MDF tool flow.

frequency (e.g., 100 MHz), and the synthesis tool to be used (currently, Synopsys Synplify or Xilinx XST are supported). On a query, MDF creates a Verilog module wrapper which instantiates the target operator. Then, MDF executes a third party tool chain, which synthesizes the wrapper and the instantiated module (using the selected synthesis tool) and then maps, places, and routes the design for the target FPGA, finally analysing the timing. At this point in time MDF is currently configured for the Xilinx ISE tool chain (synthesis → ngdbuild → map → par → trce), but can easily be adapted to other vendors' tool chains. There are *two* tool chain passes, the first pass determining the required area from the map report (a text file output by the map tool), while the second pass extracts the maximum frequency from the timing report (the output of the trce tool). Therefore, the first pass ends with the mapping step.

Two passes are needed because for the determination of timing results, another operator wrapper must be used in which all I/Os (except CLK and RESET) are buffered in a register: Logic paths not containing registers are not considered by the default period analysis of the Xilinx trce tool. Area meta data measurements must be performed without these registers, since they do not belong to the module internals.

The meta data extracted from the reports are transferred in a reply to the compiler. MDF holds queries and replies in a cache file, so that meta data which has been determined already on a query is re-used as reply for subsequent identical queries.

### 7.2.1. Measurements

Figs. 7.2 through 7.8 show area and timing data determined by MDF for a Xilinx Virtex-5, xc5vfx70t FPGA, speed grade -1, target frequency 100 MHz, using Synopsys Synplify Premier DP 9.6 for synthesis and the Xilinx ISE 10.1 tools for map, place, and route. However, when interpreting the numbers gathered with MDF, we first have to admit that these numbers do not account for effects occuring when multiple Modlib modules are used in bigger designs. On the one hand the occupied area may increase because of logic or register duplication coping with high fanouts or target frequencies. On the other hand the area may decrease because of boundary optimization. Regarding the frequency, high fanouts or complex routing can lead to an overall frequency which is lower than the lowest frequency determined by MDF.

Second, timing measurements for simple, combinatorial modules (such as a 32 bits wide adder) often heavily depend on the placement of input and output

buffer registers relative to the operator logic. The routing delay to such a buffer can be a significant part of the critical path.

Third, MDF measurements are highly dependent on the synthesis tool. For example, the tool decides if storage elements are implemented as slice flip-flops or as distributed RAM in look-up tables (LUTs). Similarly, multiplication operators can be implemented using LUTs or embedded DSP blocks.

All in all, MDF measurements should be regarded as an *estimate* for the required area and achieved frequency of Modlib operators in a larger design.

Fig. 7.2 shows measurements for different operators with invariant Modlib parameters. add (addition), bitand (bitwise and operator), and mul (multiplication) consume between 22 and 36 FFs and 41 LUTs, while the divider optimized for high throughput goes far beyond that with 3,184 FFs and 4,228 LUTs (an area optimized version significantly reduces the required resources, see Fig. 7.8). The mul operator has been implemented using three embedded DSP blocks of the Virtex-5 FPGA. Measured frequencies are between 137 and 236 MHz (note the 10 MHz unit).

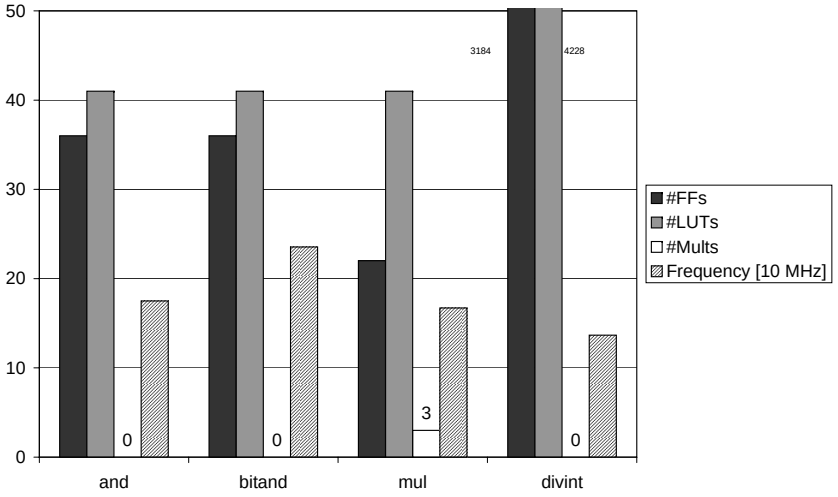


Figure 7.2.: MDF: Different operators; WA=WB=WR=32, Depth=1, QDepth=TDepth=0, StaticCT=0, StartCtrlIn=0, Sign=0; divint: AreaNSpeed=0.

Fig. 7.3 compares area requirements and frequencies of adders with different bitwidths. While the number of LUTs increases almost linearly with the

bitwidth, the frequency drops from about 400 MHz (8 or 16 bit adder) to just under 300 MHz (32 or 64 bit adder). However, the latter effect is attributable more to I/O buffer routing delays than to the longer carry chain.

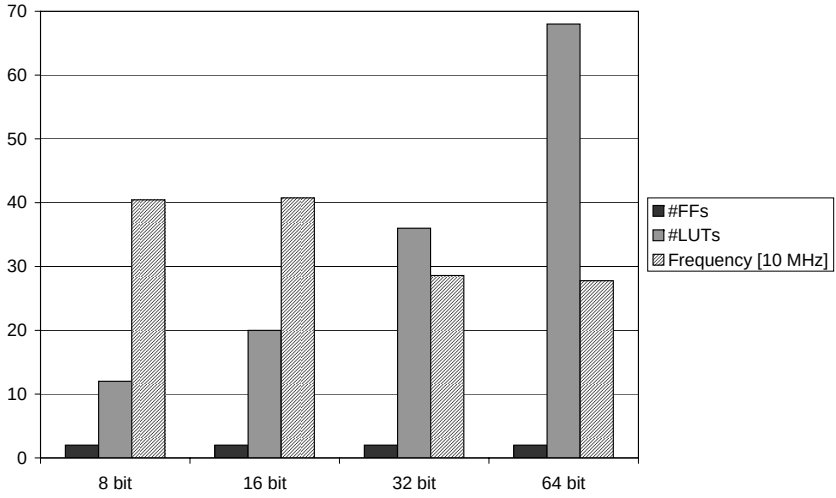


Figure 7.3.: MDF: add operator: different bitwidths; Depth=QDepth=TDepth=0, StaticCT=0, StartCtrlIn=0, Sign=0.

Fig. 7.4 compares the area requirements of fixed-latency buffers (Depth) to transparent buffers (QDepth). For buffer sizes greater than one, transparent buffers occupy fewer area resources than fixed-latency buffers (for Virtex-5, the FF:LUT reation is 1:1), the latter exhibiting a better balance between FFs and LUTs. The Depth LUT and FF curves indicate that the synthesis tool uses distributed RAM for buffer sizes greater than two.

Fig. 7.5 shows the area increase of predication due to the additional token queue and the computation logic for the acknowledge signal StartCtrlAck. The measured frequencies differ by about 30 MHz.

A comparison of results for static and dynamic CT configurations is shown in Fig. 7.6. Static CTs achieve a slightly smaller area and a much higher frequency (307 MHz static vs. 175 MHz dynamic) than dynamic CTs, due to the token propagation logic which is only needed in the dynamic case.

Fig. 7.7 shows the area increase due to greater transparent data buffers (QDepth). The greater hop from QDepth=0 to QDepth=16 is due to the buffer administration logic, which is optimized away for buffer size 0.

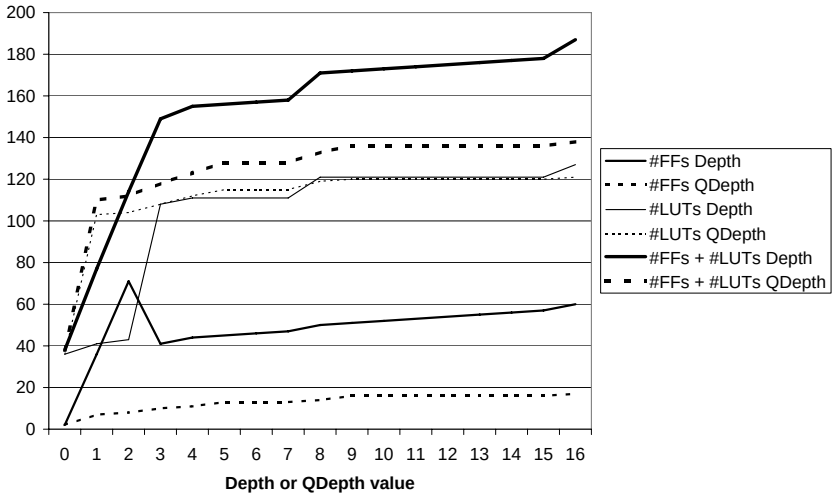


Figure 7.4.: MDF: add operator: comparing area for Depth vs. QDepth; WA=WB=WR=32, TDepth=0, StaticCT=0, StartCtrlIn=0, Sign=0.

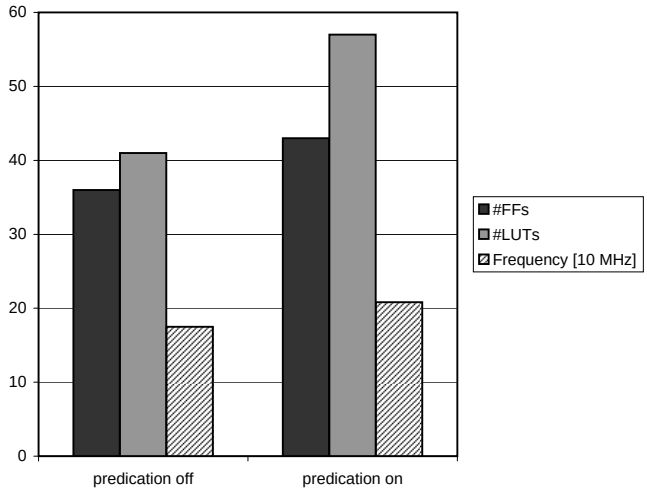


Figure 7.5.: MDF: add operator: predication (StartCtrlIn=1 vs. no predication (StartCtrlIn=0); WA=WB=WR=32, Depth=1, QDepth=0, TDepth=16, StaticCT=0, Sign=0.

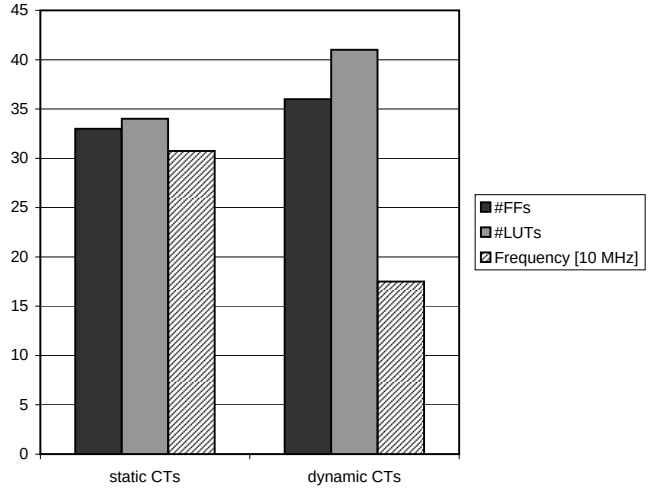


Figure 7.6.: MDF: add operator: static CTs (StaticCT=1 vs. dynamic CTs (StaticCT=0); WA=WB=WR=32, Depth=1, QDepth=TDepth=0, StartCtrlIn=0, Sign=0.

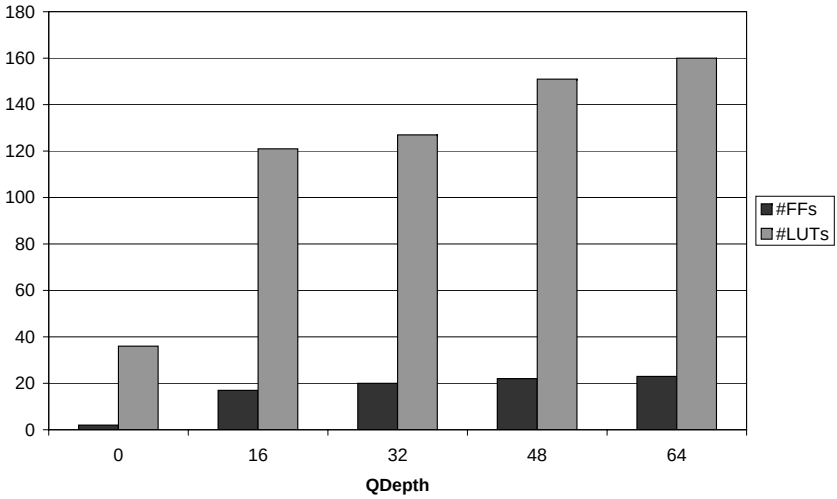


Figure 7.7.: MDF: add operator: variation of the QDepth parameter; WA=WB=WR=32, Depth=TDepth=0, StartCtrlIn=0, StaticCT=0, Sign=0.

Fig. 7.8 shows the influence of the AreaNSpeed parameter for the divider operator. The area optimized version – here, the initiation interval (II) is 8 – requires 545 FFs and 357 LUTs, while the throughput optimized version (II=1) occupies 3,184 FFs and 4,228 slices.

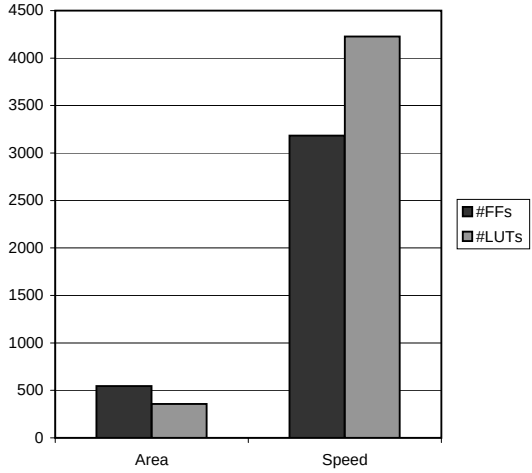


Figure 7.8.: MDF: divint operator: area optimized (AreaNSpeed=1) vs. throughput optimized (AreaNSpeed=0); WA=WB=WR=32, Depth=1, QDepth=TDepth=0, StartCtrlIn=0, StaticCT=0, Sign=0.

# 8. COMRADE 2.0 Compile Flow

Fig. 8.1 shows the COMRADE 2.0 compile flow which improves and extends the COMRADE 1.0 flow (Section 3.2). Most of the extensions are related to the back-end; however some adjustment of the front-end was necessary, too. The major changes of the SUIF2 passes are denoted by an asterisk.

At the very beginning of the flow a new goto removal pass [Webe08] transforms the input program into a t-structured program (cf. page 24). For this we have adopted the method proposed by Erosa and Hendren [ErHe94], which replaces goto statements with loops and if/else structures, and have extended the approach so that breaks and continues are replaced as well. An exception are the breaks in switch statements which do not violate the structuredness. An example of this is shown in Fig. 8.2. Here the breaks are used to prevent a fall-through to the next switch case. Finally, the pass normalizes do-while loops to while loops to actually produce a t-structured program which processes loop headers before the loop body.

The loop duplication pass of COMRADE 1.0 has been removed in 2.0 due to lack of infrastructure supporting its use in practice. We do not choose between hardware and software implementations at runtime, and instead determine the (constraint-based) partitioning at compile time – COMRADE 1.0 did not offer any method to determine such decisions at runtime anyway.

As Section 5.3.2 points out, memory forwarder nodes in the CMDFG are required for correct token flow along memory edges. The memory forwarder statements from which those nodes are created are inserted using the new memory forwarder insertion pass. This pass also makes sure that for each loop header the exit successor CFG node (the successor which is executed when the loop ends) contains a memory access (MA), which is required for the subsequent memory edge insertion. If no MA is a priori present, a memory forwarder statement is inserted here.

Instead of GLACE, COMRADE 2.0 uses the new Modlib library (Chapter 7) for the implementation of the C operator semantics in hardware; accordingly, MDF (Section 7.2) is used to extract meta data. MDF simplifies many algorithms and data structures because it is a C++ library, while GLACE is pro-

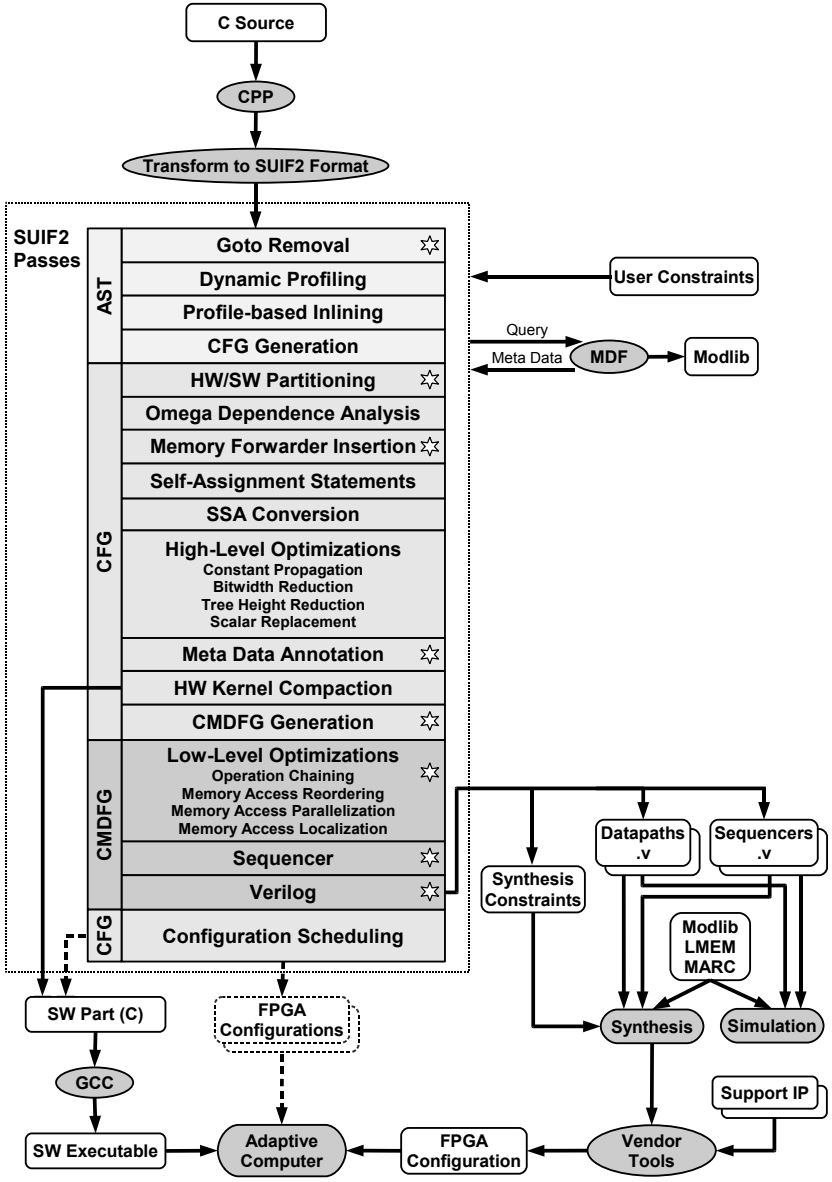


Figure 8.1.: The COMRADE 2.0 compile flow.

(a)

```
switch(c) {
  0:       a = 7;
           break;
  1:       a = 8;
           break;
  default: a = 9;
           break;
}
```

(b)

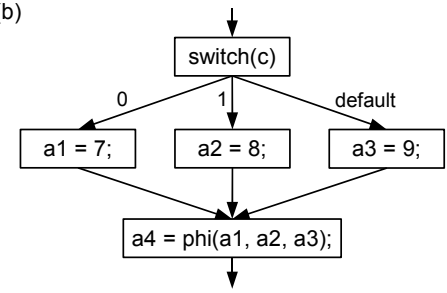


Figure 8.2.: Break statements not violating the structuredness; (a) C code, (b) CFG.

grammed in Java and so introduced a considerable overhead in COMRADE 1.0 due to the required C++/Java interfaces.

The original COMRADE 1.0 CDFG generation pass has been partly re-used (node and data edge creation) and extended with new functionality (regarding control edges and memory edges) to form the new CMDFG generation pass which implements the hardware generation concepts described in Chapter 5. The implementation of these new concepts are the key for the broad language support offered by COMRADE 2.0, e.g., arbitrarily nested loops and conditions (including pointer accesses in loop nests) which were not correctly supported yet by COMRADE 1.0. Details on how the CMDFG is built from the CFG follow in Section 8.1.

New features of COMRADE 2.0 include four low-level optimization passes which we detail in Chapter 9.

The sequencer generation pass implements the token flow concepts defined with COCOMA and represents one of the most important contributions of this work.

The Verilog creation pass has been completely reworked in COMRADE 2.0. The sequencers are written directly in Verilog instead of SLIF, because the synthesis tools today are sophisticated enough to render additional logic optimization techniques virtually unnecessary. Furthermore, the naming of wires connecting operator instances and sequencer logic is now considerably more intuitive and context-based, simplifying debugging processes. The generated synthesis constraints allow the fully automatic synthesis of the created HW kernels. The new LMEM framework provides access to local on-chip memory, increasing the available memory bandwidth considerably. However, the local

memory features (cf. Chapter 9) have been added only during the final development phase of this work, and are not completely integrated into the automatic flow. Thus, some user intervention is required for their use.

A step which is still missing in order to complete the desired framework is the integration of the existing configuration scheduling technique. For this, an adequate adjustment of the output C code needs to be implemented, including code segments for the reconfiguration of the RCU. However, a significant improvement over COMRADE 1.0 here is that through setting user constraints, multiple HW kernels can be selected and automatically synthesized into a single FPGA configuration.

## 8.1. Building the CMDFG

The transformation from the control flow-oriented, high-level CFG to the low-level, data flow-based CMDFG representation is a central part of the COMRADE 2.0 hardware back-end. We will first explain the CMDFG creation on a step-by-step basis: CMDFG nodes (Section 8.1.1), data edges (Section 8.1.2), control edges (Section 8.1.3), and memory edges (Section 8.1.4). Note that each of these steps operates on a HW region within the CFG, i.e., a CFG can contain several HW regions, from each of which we generate a separate CMDFG. Second, we investigate the correctness of the approach (Section 8.2) and finally consider an alternative token flow for nested conditions in Section 8.3.

### 8.1.1. Nodes

To create CMDFG nodes for a CFG HW region, the statements in each HW region node are traversed. From each of these statements one or more CMDFG nodes are generated according to the complexity of the contained expressions. An example is shown in Fig. 8.3, which already includes CMDFG data edges for easier correlation with the C code; however, the data edges are actually inserted in a later step. For `c = a + b`, only a single adder is created. From `*p = a - b + c`, the compiler builds a `store` node with an address input for `p`, a subtraction operator, and an adder.

Appendix B shows a complete list of assignments from C operations and statements to CMDFG node types.

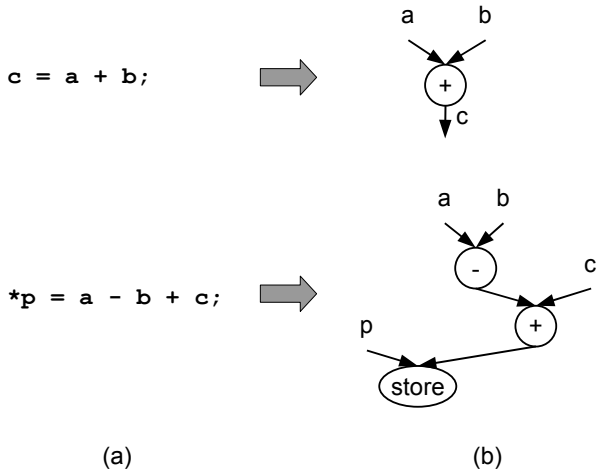


Figure 8.3.: (a) Statement contained in a CFG node; (b) generated CMDFG nodes and data edges.

### 8.1.2. Data Edges

The C statements and expressions in the CFG nodes define producers and consumers of variables. A **producer** of a variable is an operator which computes a value that is assigned to that variable; a **consumer** is an operator reading the variable value. For example, given the two statements `c = a + b; d = c - 1`, the addition in the first statement is a producer of `c`, while the subtraction in the second statement is a consumer of `c`. Obviously, consumers are data dependent on producers; these dependences are modeled by data edges. As the CFG has already been transformed to the SSA form, each variable has exactly one producer. Thus, inserting data edges is done by iterating over the producers and for each producer adding a data edge to each of its consumers[1].

When a data edge is connected to a consumer, it is important to choose the correct input port, because CMDFG nodes (as well as C operators) are generally not commutative. For that purpose, the operator input computed first in the program flow is always connected to data port `A` of the CMDFG node[2].

[1] Connecting a data edge to a producer or consumer means to connect it to the CMDFG node that has been generated from the producer or consumer operator.

[2] For muxes, the least significant bits of the A port connect to the first input according to the program flow, etc.

Bitwidths are assigned to the data edges according to the C data types.

Self-assignment statements (n = n) produce cyclic data paths of length one, where the output of a loop mux directly connects to the continue input of the same loop mux. This is problematic for dynamic scheduling, because certain conditions can lead to deadlocks. Such a situation is shown in Fig. 8.4(a). The loop mux, currently having an AT at its output, must not accept another incoming AT from a data predecessor before the output AT has been consumed by all data successors. As the loop mux is a successor of itself, it will not accept any further incoming AT (deadlock). To solve this problem we expand such data path cycles through insertion of a nop node (Fig. 8.4(b)). An alternative would be to set the QDEPTH parameter of the loop mux to at least one, allowing it to consume another AT via the continue input *before* the current output has been acknowledged.

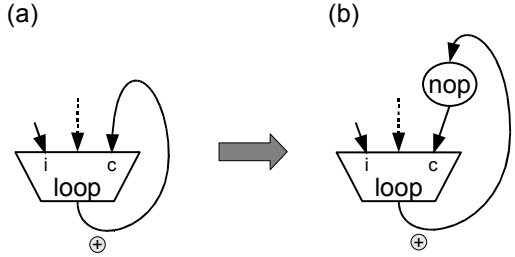


Figure 8.4.: (a) Before, (b) after nop node insertion for data flow loops.

### 8.1.3. Control Edges

Control edges are used in the CMDFG for different purposes (cf. Section 5.3.2). For example, they implement the predicated execution of non-speculative operations as well as the selection from a set of alternative data paths. Furthermore, they are used to forward tokens in control hierarchies, and to control the data propagation from loop muxes to outer loop levels. Due to these diverse application domains the control edge generation is subdivided into several sub-steps as shown in Algorithm 1. In the following we detail each of these steps.

**Algorithm 1** Insertion of control edges.

1: insert control edges to mux inputs
2: insert control edges to loop mux inputs
3: insert control edges to loop mux outputs
4: insert control edges to constants
5: build control hierachies
6: insert control edges to memory accesses
7: loop control nets
8: HW/SW interfaces
9: merge control edges

### Mux Inputs

Alternative data paths originating from if/else or switch branches are joined at muxes. The preliminary objective here is to insert a control edge to each data predecessor of the mux. Only if the control condition is true, is the result of that predecessor forwarded to the mux; otherwise that (mis-speculated) piece of data is discarded. In this preliminary model the mux simply accepts the active input, where not more than one input may be active at a time. (In Step 9, the control edges to mux predecessors will be merged, which will drop the latter requirement.)

Fig. 8.5 shows an example in which control edges are inserted for the predecessors of mux $m$. To find the **CMDFG mux control node** for $m$ (i.e., the node computing the condition that decides which input of $m$ is chosen), we first determine the CFG node $m_{cfg}$ that contains the phi statement from which $m$ originates (1.). Obviously, $m_{cfg}$ is a join node. Due to the structuredness there is exactly one branch node $c_{cfg}$ for $m_{cfg}$ (2.). We find $c_{cfg}$ by inspecting the predecessors of $m_{cfg}$. If such a predecessor is a branch node, that node is already $c_{cfg}$. Otherwise an arbitrary predecessor $p_{cfg}$ is chosen; the controller of $p_{cfg}$ is $c_{cfg}$. Such a controller $c_{cfg}$ exists, because $p_{cfg}$ is an alternative branch in an if/else or switch body. There is only one such controller because of Proposition 2.1.1. Having $c_{cfg}$, the CMDFG control node $c$ is the node which has been created from the condition in the last statement of $c_{cfg}$ (3.).

Now we insert a control edge $(c, p_i)$ for each mux predecessor $p_i$. For this we have to extract the control annotation (==0 or ==1 in Fig. 8.5(b)) that defines for which output of $c$ the value computed by $p_i$ is forwarded to $m$. Here, the first step is to determine if the CFG node $p_{i,cfg}$ from which $p_i$ originates is dominated by $c_{cfg}$. If so, $p_{i,cfg}$ is obviously located in an alternative branch

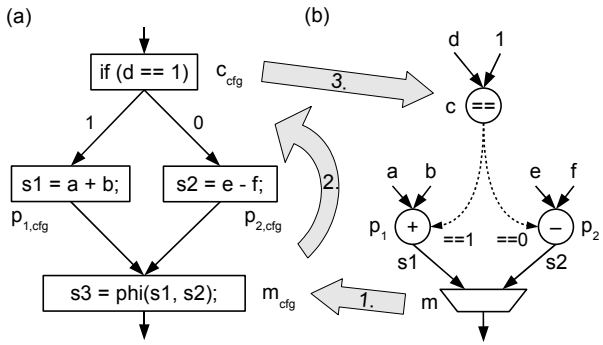


Figure 8.5.: (a) CFG, $p_{i,cfg}$ dominated by $c_{cfg}$; (b) CMDFG with control edges inserted for mux predecessors.

between $c_{cfg}$ and $m_{cfg}$. We then find the successor of $c_{cfg}$ which dominates $p_{i,cfg}$ (in Fig. 8.5(a) this is $p_{i,cfg}$ itself); the value annotated at the CFG edge from $c_{cfg}$ to that successor (0 or 1 in the example) is used for the control edge annotation[3].

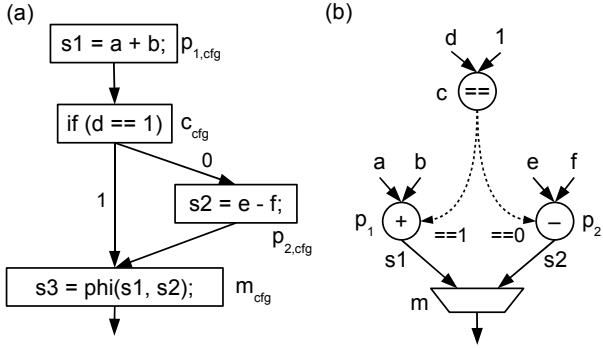


Figure 8.6.: (a) CFG, $p_{1,cfg}$ *not* dominated by $c_{cfg}$; (b) CMDFG with control edges inserted for mux predecessors.

The example in Fig. 8.6 shows a case in which one predecessor of $m_{cfg}$, $p_{1,cfg}$ is *not* dominated by $c_{cfg}$; instead, $p_{1,cfg}$ is located before $c_{cfg}$ in the program

[3] An exception is the default annotation of switch statements, in which the actual annotation for the CMDFG control edge is computed from the remaining outgoing CFG edges of the CFG controller node.

flow. According to the following Lemma 8.1.1 there can only be one such non-dominated node $p_{i,cfg}$. So we first extract the annotations for the dominated nodes and mark the associated successors of $c_{cfg}$ as used. For if/else there can be at most one unused successor, while a switch may have several. We combine the annotations of the edges from $c_{cfg}$ to the unused $c_{cfg}$ successors and assign the resulting control annotation to the control edge $(c, p_i)$.

**Lemma 8.1.1.** *Given a t-structured CFG with a phi statement ps in a non-header join node $m_{cfg}$ and an associated branch node $c_{cfg}$. There can only be one $m_{cfg}$ predecessor $p_{i,cfg}$ which is not dominated by $c_{cfg}$.*

*Proof.* Assume there are two such predecessors $p_{i,cfg}$ and $p_{j,cfg}$, $i \neq j$. *ps* w.l.o.g. computes $var = phi(var_1, \ldots, var_i, \ldots, var_j, \ldots, var_n)$, where $var_i$ is defined in $p_{i,cfg}$, and $var_j$ is defined in $p_{j,cfg}$. $p_{i,cfg}$ and $p_{j,cfg}$ are located in alternative branches, i.e., every path from one to the other contains a join node, because otherwise $var_i$ and $var_j$ would not be inputs of the same phi statement. As $p_{i,cfg}$ and $p_{j,cfg}$ are both not located on a path between $c_{cfg}$ and $m_{cfg}$ (because they are not dominated by $c_{cfg}$), there must be a join node in the program flow before $c_{cfg}$, which contains a phi statement merging at least $var_i$ and $var_j$ to a new defined variable $var_k$. But this means that $var_i$ and $var_j$ cannot be inputs of *ps*, which must read $var_k$ instead of $var_i$ and $var_j$. □

### Loop Mux Inputs

A loop mux is a data path connector between an outer and an inner loop. It has two data inputs; the init input is connected to the outer loop data path, and the continue input receives values from the body of the inner loop. In the example in Fig. 8.7(b) the loop mux is initialized with a constant four, while the continue input connects to the adder which increments i once per inner loop iteration. The control edge for the continue input is created similarly to mux predecessors, where finding the controller node (the < node in the Figure) is much easier here, because it originates from the same CFG node (highlighted in Fig. 8.7(a)) as the loop mux. In the example, the control edge (<, +) is inserted with annotation ==1.

For the init input from the outer loop, we do not insert any control edge. This is because an outer loop data path already delivers one data word per outer loop iteration, which is the desired behavior here. However, note the activation of constants below (Section 8.1.3); for the example in Fig. 8.7, that step will insert a control edge to activate the const node 4.

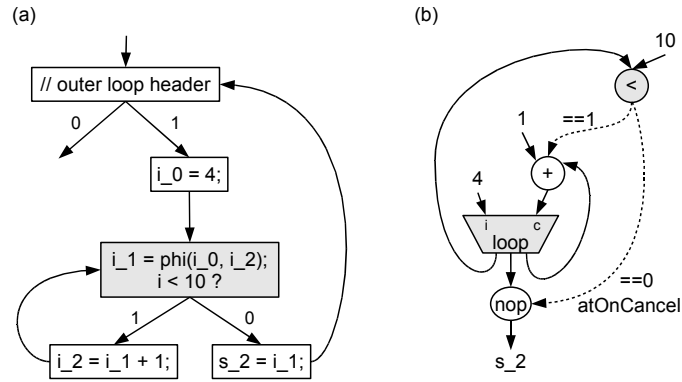


Figure 8.7.: (a) CFG section; (b) CMDFG with control edge inserted for the loop mux continue input and its output; CMDFG nodes with gray background originate from the CFG node with gray backgound in (a).

### Loop Mux Outputs

To insert the loop mux output control edges illustrated in Section 5.3.2 (Loops) we first determine for a given loop mux the successor nodes which originate from a CFG node of the outer loop. For each such successor $s$, we insert a control edge[4] from the loop control node with two annotations: ==0 and atOnCancel (as illustrated in Section 5.3.2). An example is the control edge $(<, nop)$ in Fig. 8.7.

### Constants

To activate const nodes, we simply iterate over the CMDFG nodes and insert a control edge from the always node to each const node.

### Control Hierachies

We iterate over the CMDFG nodes. For a node $n$ which computes an if/else, switch or loop condition, its **CMDFG controller node** (i.e., the node control-

[4] If s is already targeted by another control edge, a nop node is inserted between the loop mux and s, and the control edge is attached to that new nop node.

ling the execution of $n$) is found by again using the CFG control dependence relation (cf. Fig. 8.8). First, we locate the originating CFG node $n_{cfg}$, then determine the controller $c_{cfg}$ in the CFG and finally use its last statement to locate the sought-after CMDFG control node $c$.

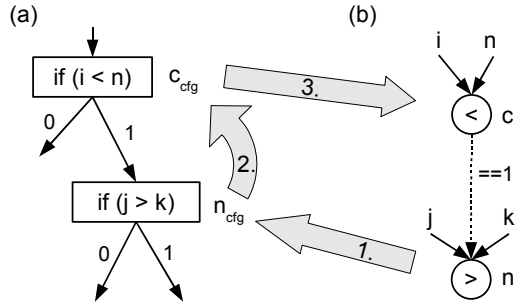


Figure 8.8.: (a) CFG section; (b) CMDFG with a control edge inserted between nested conditions.

An exception is the loop condition of the outermost loop, which has no controller in the current HW region. Here, we use the initial node of the corresponding SW/HW transition as the control edge source. This establishes that the SW/HW transition actually occurs before a memory access inside the HW region is started and before an IRQ is signaled by the hardware.

The loop condition reactivating any nodes (cf. Section 5.3.2, Loops) are inserted for each loop condition[5].

### Memory Accesses

This step adds control edges to the non-speculative memory access (MA) nodes: load, store, and mf. To create the nCT and nAT annotated edges motivated in Section 5.3.2 (Nested Loops), we again first determine the controller in the CFG, i.e., the CFG controller node of the CFG node from which the CMDFG MA node has been created. In Fig. 8.9, MA1 and MA2 have been created from CFG node $m_{cfg}$, the controller of which is $c_{cfg}$. If an MA is the first MA in its CFG node (basic block) according to the program flow (MA1 in the Figure) and the CFG node ($m_{cfg}$) has a loop header as its direct predecessor ($l_{cfg}$), the compiler inserts an nCT control edge from the CMDFG node comput-

[5]This will be refined for nested loops below.

ing the loop condition to the CMDFG MA node, annotated with the loop exit condition ==0 (edge (>, *MA1*) in the Figure). Furthermore, a nAT control edge is inserted from the MA controller CMDFG node (reusing the control dependence relation on the CFG) to the MA node (edge (<, *MA1*)). For successive MAs, nCT/nAT splitting is not necessary, because the memory edges which will be added later already avoid a premature MA execution. Instead a standard control edge (i.e., without nCT or nAT) is then inserted (as edge (<, *MA2*)).

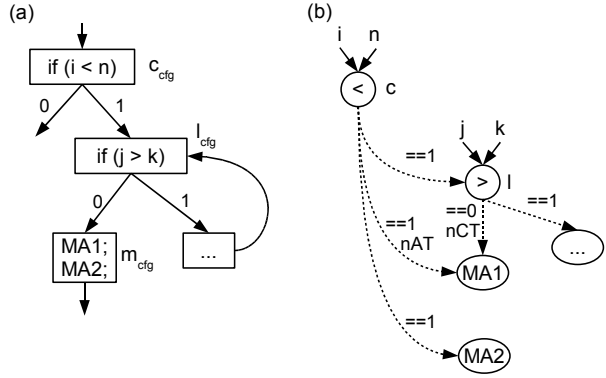


Figure 8.9.: (a) CFG section; (b) CMDFG with control edge inserted for memory accesses MA1 and MA2.

### Loop Control Nets

Here, we furnish the existing control hierarchies with control edge structures as shown in Fig. 8.10 to make sure that an outer loop condition is not reactivated before all inner loops have finished. To ensure this, loops are processed sequentially from inner to outer levels. Conditions at the same level are combined with an all node because all these loops have to finish before the outer loop may execute the next iteration. In the example in Fig. 8.10, $j > m$ and $k == p$ must both be false (== 0), before $i < n$ may be reactivated. This step may require that a preliminary any node destined for loop reactivation is removed.

Conditions on different levels are combined with an any node as illustrated by the example in Fig. 8.11.

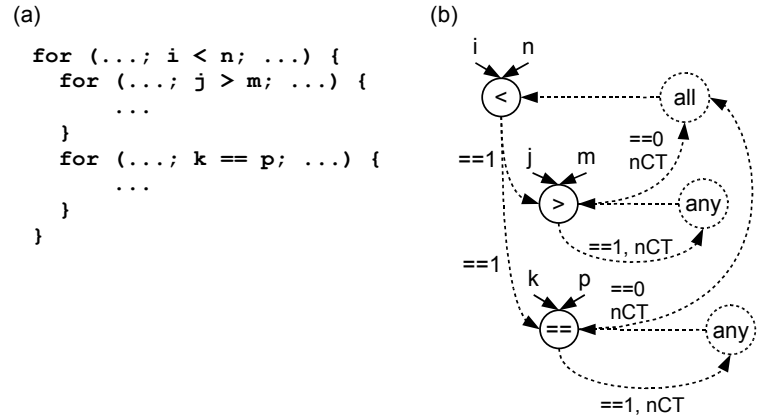


Figure 8.10.: (a) C code, (b) generated CMDFG control net.

### HW/SW Interfaces

For each HW/SW transition a const node is created, its value representing the CFG node number of the HW exit node (i.e., the node which is the source of the HW/SW transition). Fig. 8.12 shows an example comprising two HW/SW transitions in one HW region. Furthermore, for each variable which is transferred from HW to SW, a control edge from the associated outreg to the const is inserted; in the case of more than one outreg, an and node combines the edges as shown in the Figure. If there is only one outreg, the and node is omitted. If there is more than one HW/SW transition in the current HW region an irqDataMux combines the IRQ exit codes as shown in Fig. 8.12(b); if there is only one the const node is connected directly to the irqreg.

### Control Edge Merging

In this last step, the preliminary control edges to mux predecessors are replaced by control edges which target the mux directly (cf. Fig. 8.13)[6]. This representation is even closer to the target hardware; the control edge now directly influences the select signal of the multiplexer.

A similar replacement is made for control edges pointing to loop mux prede-

[6]Using preliminary control edges allows for a more generic control edge handling while building the CMDFG, and thus simplifies the implementation.

(a)

```
for (i = 0; i < n; ++i)
  if (i == 4)
    for (j = 0; j < m; ++j)
      ...
```

(b)

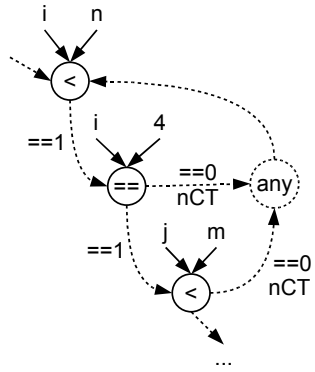


Figure 8.11.: Nested loops: (a) sample code snippet, (b) generated COCOMA section.

cessors. Here, however, only the continue input predecessor is affected by the edge replacement, because the init input is not controlled by the associated loop condition node. As a result the semantics of control edges to a loop mux are slightly different from mux control edges. While an edge pointing to the latter affects all data predecessors, a control edge to a loop mux affects only the continue input. That is, a loop mux is initialized without consuming an AT from the control predecessor; similarly, a false condition in the control predecessor will create a CT for the continue predecessor of the loop mux, but not for its init predecessor.

### 8.1.4. Memory Edges

For the implementation of memory accesses COMRADE 2.0 assumes that a HW kernel has access to a memory back-end supporting cached accesses to the main memory, as planned in Section 5.1. The cache is accessible via a bus which allows fast single word transfers. The HW kernel is provided a cache port to access that bus. Note that this is the very basic memory system supported by COMRADE; extensions are presented in Chapter 9.

COMRADE uses memory edges to enforce the program order sequence of memory accesses (as defined by the source C code). This is important to adhere to write-after-read (WAR), read-after-write (RAW), and write-after-write (WAW) dependences on the one hand, and to prevent simultaneous cache port

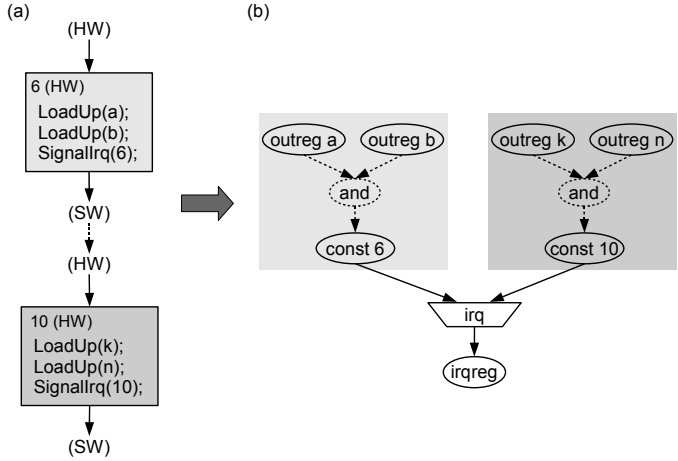


Figure 8.12.: (a) CFG section with two HW/SW transitions, (b) generated CMDFG subgraph for IRQ signaling.

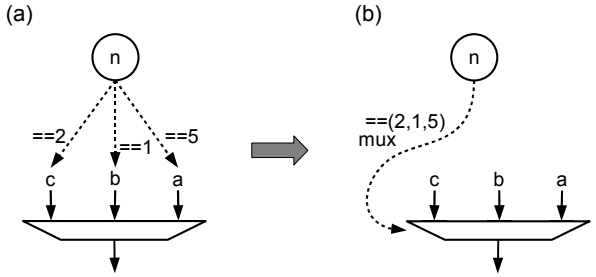


Figure 8.13.: (a) Before, (b) after control edge merging.

**Algorithm 2** Construction of memory edges in program order

```
 1: for all CFG nodes n_cfg (except SW services) containing a memory access do
 2:     insert memory edges between the accesses located in n_cfg in program order
 3:     lms_cfg := last memory statement in n_cfg
 4:     lma := CMDFG memory access node generated from lms_cfg
 5:     if n_cfg is a branch node then
 6:         c := CMDFG condition node computing the branch condition of n_cfg
 7:         add memory edge from lma to c
 8:     else
 9:         {the successor succ_cfg of n_cfg is a branch or a join node}
10:         fma_cfg := first memory access statement in succ_cfg
11:         fma := CMDFG node created from fma_cfg
12:         add memory edge from lma to fma
13:     end if
14: end for
```

accesses on the other hand; for the sake of the latter, even memory edges from load to load are required.

Algorithm 2 shows how COMRADE 2.0 inserts memory edges. Step 2 inserts memory edges for accesses located in the same CFG node. For branch nodes, the last memory access lma in the node is connected to the branching condition c (Steps 6-7) to guarantee the program order for dependent accesses in branch targets, i.e., c is evaluated after lma is done, hence accesses in the branch target (controlled by c) are not started prematurely. Steps 10-12 insert memory edges from the last memory access of a non-branch node to the first access of the successor node, obeying inter-CFG node memory dependences. This also closes the memory dependence chain across the loop header, so that all memory accesses of the current loop iteration are done before the first access of the next iteration starts.

## 8.2. Correctness

In this work we examine basic concepts for the creation of combined HW/SW systems from C. We do *not* intend to prove the correctness of all suggested approaches, which would definitely exceed the scope of this work. However, we do allude some correctness hints, through representative tests on the one hand, and by analyzing basic principles of the token flow on the other hand.

We have run COMRADE 2.0 on a set of synthetic benchmarks, covering different combinations of (nested) conditions and loops – with and without memory accesses – and testing different parameter settings (e.g., dynamic vs. static CTs

and different buffer sizes). The generated HW kernels have been simulated to observe the correct behavior. To be able to reproduce the tests, their C source codes are given in Appendix C, together with simluation results.

Before we analyze the token flow principles further, we will define three terms helpful for the discussion.

**Definition 8.2.1.** Given a t-structured CFG with node set $N$, edge set $E$, and a distinct loop header $l$.

The **explicit loop body** of $l$ is a subset $ELB(l) \subseteq LB(l)$ of the loop body of $l$, so that each node in $ELB(l)$ is control dependent on $l$.

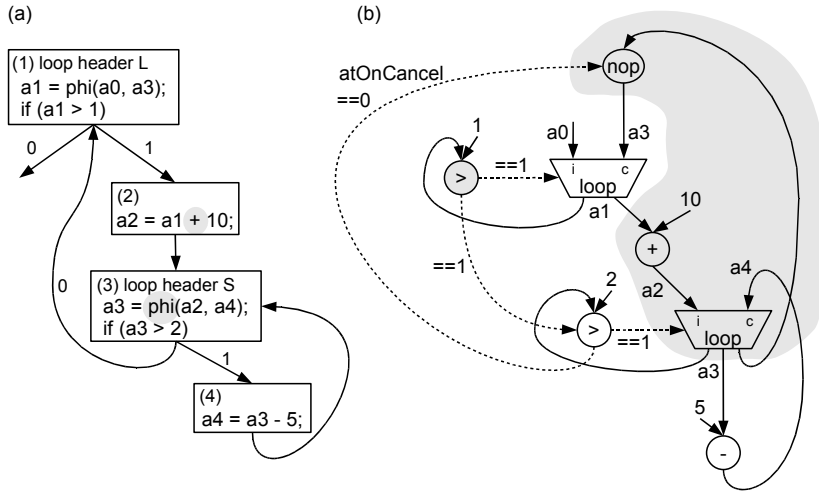


Figure 8.14.: (a) CFG section, (b) generated CMDFG subgraph.

As an example, in Fig. 8.14 nodes 1 and 3 are loop headers, $\{2, 3, 4\}$ is the loop body of loop header 1, $\{4\}$ is the loop body of 3, $\{2, 3\}$ is the explicit loop body of 1, and $\{4\}$ is the explicit loop body of 3. Obviously, explicit loop bodies do not overlap, which allows the definition of loop levels based on explicit loop bodies.

**Definition 8.2.2.** Given a t-structured CFG containing a HW region $H$ with node set $N$ and edge set $E$.

The **loop level function** assigns each node in $H$ a non-negative integer, so that the level of an explicit loop body equals the level of its header increased by 1,

i.e., $LL : N \to \mathbb{N}_0$,

$$LL(n) := \begin{cases} LL(h) + 1 & \text{if } n \text{ has a loop header } h \text{ in } H, \\ 0 & \text{otherwise.} \end{cases}$$

Assuming that all nodes in Fig. 8.14 are contained in a HW region $H$, and node 1 is the outermost loop header in $H$, then $LL(1) = 0$, $LL(2) = 1$, $LL(3) = 1$, and $LL(4) = 2$.

According to their origin in the CFG, CMDFG nodes are associated to exactly one explicit loop body and therefore have the same loop level. The association of CMDFG nodes to CFG loop levels drives the definition of loop data paths.

**Definition 8.2.3.** Given a t-structured CFG containing a HW region $H$ with node set $N$ and edge set $E$, and a CMDFG generated from $H$.

A **loop data path (LDP)** is a set of CMDFG nodes weakly connected through data edges, so that each node in the LDP has at least one data input and one data output and all nodes in the LDP have the same loop level. Furthermore, an LDP is maximal, i.e., there is no LDP which is a proper superset of a given LDP.

As an example, the shaded region on the far right in Fig. 8.14(b) surrounds an LDP consisting of a nop, an adder, and a loop mux node, having the same loop level 1. Although the nop node has no equivalent CFG node, it has loop level 1 because it is simply a buffer for a3, which is defined at that loop level. The gray-colored > node computes the loop condition for the LDP (which itself is not part of that LDP). In Fig. 8.14(a) the operations from which this LDP originates have been marked with gray background.

For each variable which is read or written in an explicit loop body, its loop header contains a phi statement (after SSA conversion) from which a loop mux is created in the CMDFG. These loop muxes can be understood as connectors between LDPs of different loop levels.

By analyzing the token flow across LDPs, especially the synchronization of data, control and memory flow, we will now illustrate (though not prove) that no deadlocks occur in an LDP. Fig. 8.15 shows a generalized LDP of level $k$, where loop_c is the associated loop control node (a). Such a LDP can have data inputs from const nodes (b), or from loop muxes (c) of loop level $k - 1$ (originating from the same CFG node as the loop condition loop_c). Possible nodes contained in the LDP include operation nodes (d) and nops (e), loads (f), muxes (g), and loop muxes (h) of level $k$. LDP data outputs can lead to operation nodes without outgoing data edges (i.e., control nodes (i)) and store nodes (j),

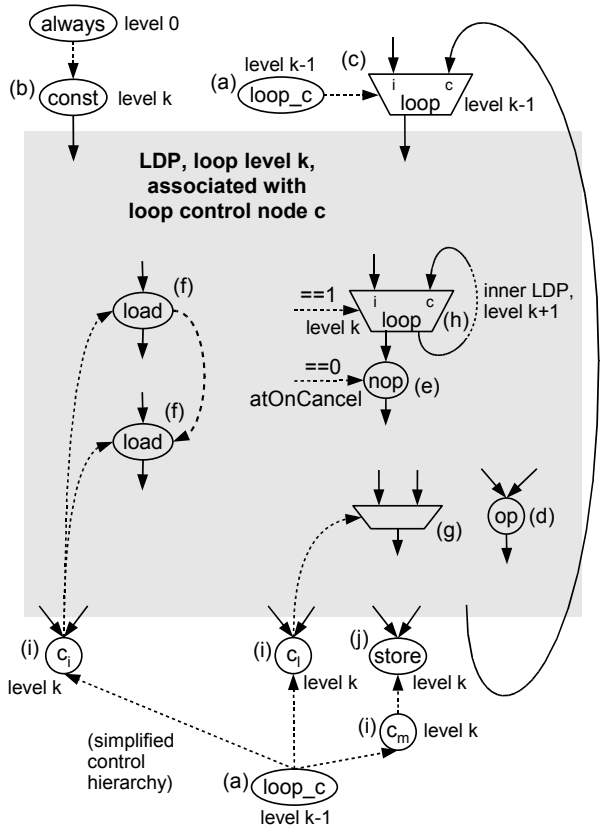


Figure 8.15.: Generalized LDP (gray background) with incoming and outgoing CMDFG edges; note that nodes not contained in the LDP can nevertheless have the same loop level.

where both have loop level $k$; the third kind of outputs are loop muxes at level $k-1$, originating from the loop header of the LDP, so that data flows into the continue input of such a loop mux (c).

### 8.2.1. Static CTs

For a given LDP a basic token flow principle of COCOMA in the static CT model is that per token (AT or CT) which enters the loop control node loop_c, an AT is propagated along each incoming and each outgoing data edge of the LDP, with the exception of data inputs from const nodes. To understand this, we analyze the I/Os and the interior of the LDP, first for the static CT model. We start off by reviewing the data inputs.

- A loop mux (Fig. 8.15(c)) receives an initial AT from an outer loop or through a SW/HW transition. We associate this initial value with the first token that enters loop_c. If loop_c is true, the value (once passed through the LDP) re-enters the loop mux via its continue input; we then associate that new value with the next token which enters loop_c. This iterates until loop_c is false. If loop_c is actually false, or the first token entering loop_c is a CT, then a CT is generated at the continue input of the loop mux, which finally eliminates the result coming in from the LDP.
- const nodes are an exception because they are connected to an always node and thus deliver a steady stream of ATs. This is however not harmful because they cannot produce erroneous results. Constant inputs are synchronized with other data inputs in operation nodes or synchronized with control inputs in muxes or loop muxes.

Thus, all inputs (except consts) deliver exactly one AT per token entry of loop_c. We now analyze the LDP interior, assuming an AT per incoming data edge.

- Load nodes (f) have an address input and a data output for the loaded value (both data edges) and an incoming control edge. The control node is either loop_c or a node $c_i$ which is a sub control node of loop_c (due to an if/else or switch in the C code). When a token enters loop_c, the control hierarchy establishes that $c_i$ will receive a token, too. If $c_i$ is active and the control condition is true, then the memory access is executed, consuming the input address via the incoming data edge and outputting the result. If $c_i$ is canceled or active with false control condition, a CT is sent to the load. Now, although no memory access is performed, the load

consumes the incoming address and provides a data output value (simply dummy data). So, in any case, the load will consume an AT from the data predecessor and produce an AT for the data successors. The outgoing *memory edge* of the load forwards an AT only if the memory access has actually been performed. In that case, the AT is consumed by the memory successor; otherwise, the memory successors do not anticipate an incoming AT, because they are canceled, too.

- A mux (g) receives exactly one AT from each of its alternative data inputs. Accordingly, it receives exactly one AT (and thus select value) from its controller node $c_l$, which is either identical to loop_c or reachable from loop_c through the control hierarchy. The mux selects an input according to the select value and propagates this to its data output. If none of the inputs is selected, a dummy value is propagated (cf. Sections 5.3.2 (Nested Conditions) and 8.3), so there will be a data output in any case.
- Loop muxes (h) of level $k$ originate from a subloop header. They are initialized once by an AT from the LPD per token that enters loop_c. That AT flows into the LDP of the subloop and re-enters the loop mux via its continue input for each subloop iteration. The controller node (not shown) of that inner loop, being again in the control hierarchy of loop_c, is reactivated for each iteration by the loop control net mechanism explained above. As soon as the inner loop ends or in the case that the inner condition is canceled, the nop node (e) is activated, delivering an AT for successors in the LDP.
- All other nodes in the LDP (having only data I/O edges) perform their operation as soon as all inputs are available and propagate an AT when their computation is finished.

In this manner, ATs are propagated from LDP inputs to LDP outputs. We finally review what happens at the outputs.

- Control nodes $c_i$, $c_l$, and $c_m$ are in the control hierarchy of loop_c and receive exactly one token per token entry of loop_c. If such a control node receives an AT via its incoming control edge, it consumes the data inputs from the LDP and creates an AT for the outgoing control edges. If a CT enters one of the control nodes, the incoming data from the LDP is consumed and discarded, forwarding a CT along the outgoing control edges.
- Store nodes (j) behave similarly, except that a memory access is performed instead of activating outgoing control edges.

- The output to the loop mux (c) has already been covered in the discussion regarding the LDP data inputs.

This concludes the discussion for the static CT model, substantiating that, per token (AT or CT) which enters a loop control node, an AT enters and finally exits the associated LDPs. This is an important result regarding the synchronization of control and data flow on the one hand, and strongly suggests the absence of deadlocks on the other[7].

In the next Subsection, we consider the dynamic CT model.

### 8.2.2. Dynamic CTs

The dynamic CT model is a bit more complex. Instead of ATs moving strictly top-down through the LDP as described for Fig. 8.15, CTs are allowed to move bottom-up, i.e., in reverse direction along data edges. The basic principle for LDPs must therefore be adapted:
*For each token (AT or CT) which enters the loop control node loop_c and for each incoming and each outgoing data edge of the LDP (except data inputs from const nodes), either an AT moves along the data edge or a CT moves in the reverse direction.*
We will however omit a detailed discussion here, suffice it to say that in the dynamic CT model, the location where token extinction takes place is variable instead of being fixed at control edge targets.

However, there is a peculiarity of the COMRADE 2.0 dynamic CT model: Nodes which are control edge targets (muxes, loop muxes, loads) do *not* accept CTs from their data successors. In these cases the CT waits in the successor node until an AT comes in (possibly carrying dummy data). This is however not harmful because in each iteration an AT enters the LDP via each data input, so that an input never waits for a CT which is blocked inside the LDP.

A complete proof for the absence of deadlocks would require to undergo an analysis of the whole COCOMA sequencer and the behavior of all COCOMA nodes (and thus Modlib operators), which is outside the scope of this work. Instead, the above observations show the plausibility of our approach and allow a more general and comprehensive view of token flow. We have tested COCOMA with many diverse examples comprising different control flow scenarios to be reasonably certain that all practically relevant cases have been covered (cf. Appendix C).

[7] Note the concluding statements of Section 8.2.2, which apply here, too.

## 8.3. Redundant Control Dependences

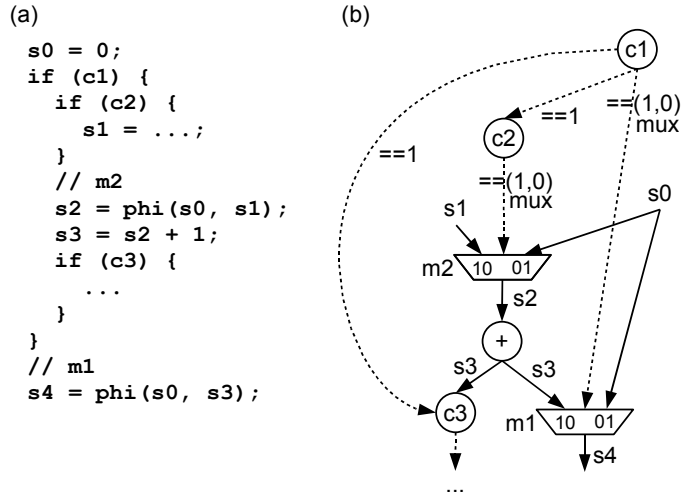


Figure 8.16.: (a) C code in SSA form; (b) generated CMDFG section.

In nested loops subconditions may be canceled, so that all of the alternative inputs of a mux are discarded. An example is shown in Fig. 8.16: If $c1$ is false, $c2$ receives a CT via its incoming control edge. Then, $s1$ and $s0$ are both discarded in $m2$. The mux outputs a pseudo AT (pAT) carrying dummy data, because the successors of $m2$ expect an AT. We have implemented an alternative approach in [GäKo08] which does not create pATs at mux outputs. Instead, we have to make sure that the successors of such muxes do not wait for an AT in the cancel case; in other words, they may not create excess CTs. The creation of CTs in the cancel case can be prevented by nCT annotations[8]. In Fig. 8.16(b) a nCT annotation of control edge $(c1, m1)$ dedicated to the $s3$ input would do the job. While this works for muxes, a nCT annotation is not possible for $(c1, c3)$, because the CT is needed by $c3$ for the propagation to its control successors. Because $c3$ expects an incoming AT along its data inputs for each iteration, a pAT must be inserted for each data input of $c3$. This can be done in COCOMA using an *alwAct* annotated control edge, which delivers an AT (ideally directly at the $c3$ data input) although the source of the edge is false or canceled.

[8] In [GäKo08], which points control edges to mux predecessors instead of using direct control edges to muxes, this is equivalent to the removal of control edges.

However, that alternative approach turned out to be much more complex and error-prone, hence we have decided to directly create pATs at mux outputs. Another advantage of the current model is the invariance presented in Section 8.2, which is essential when discussing the correctness of our approach.

# 9. Low-Level Optimizations

COMRADE 2.0 can perform several low-level optimizations on the CMDFG depending on the parameters of the actual target architecture (e.g., target frequency or speed grade). Therefore, Section 9.1 first introduces our evaluation platform ACE-M5. Section 9.2 then compares dynamic CTs to static CTs regarding runtime and resource requirements, before we present three actual low-level optimization techniques in Sections 9.3-9.5. Using the optimizations presented so far, Section 9.6 compares performance results, analyzes the current bottlenecks, and shows how they can be removed using memory localization.

## 9.1. Test Platform ACE-M5

Fig. 9.1 shows the architecture of the adaptive computer ACE-M5, which we use for our experiments. The ACE-M5 essentially consists of the Xilinx ML507 development board (kindly donated by Xilinx) and a support package provided by the Embedded Systems and Applications Group (ESA), headed by Professor Andreas Koch at the Technische Universität Darmstadt, including IP cores and a Linux operating system (OS). The Virtex-5 FPGA on the ML507 includes an embedded PowerPC processor which hosts the OS. A management PC acts as external console and also provides file sharing services to the ACE-M5 using NFS. On the ACE-M5 side, the required Ethernet IP block is connected to the Processor Local Bus (PLB). Hardware kernels generated by COMRADE 2.0 are located in the RCU. Both the PowerPC and the RCU share access to 256 MB of DDR2-SDRAM main memory via the DDR2 controller, the PowerPC using its own embedded cache (32 KB instruction, 32 KB data) connected to the memory controller interface (MCI), and the RCU using the cache provided by the MARC [LaKo00] memory access system (4 KB). Furthermore, the PowerPC has memory-mapped access to the RCU registers, bypassing the caches. The RCU can send an IRQ to the PowerPC to signal the end or interruption of a computation.

The ACE-M5 implements the AISLE architecture [LaKo09], so that the RCU can use the same virtual address as the CPU for referencing data stored in

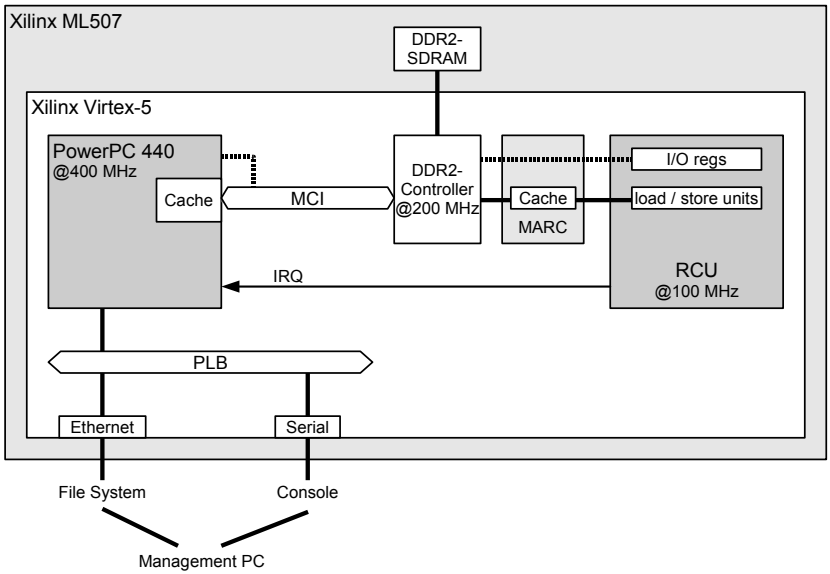


Figure 9.1.: The ACE-M5 architecture.

the main memory. This allows for passing complex data structures between software and hardware by just transferring a pointer.

The frequencies shown in Fig. 9.1 are derived from the original Xilinx ML507 reference design. Our concepts are not limited to a certain technology or target frequency; however, here we will use 100 MHz here as target frequency for our tests.

The connection between RCU and DDR2 is 128 bits wide and runs at 200 MHz[1], offering a maximum throughput of 3.2 GByte/s.

The fully associative MARC cache is configured for 32 cache lines (128 bytes each), and currently supports up to four parallel accesses (reads or writes, each 32 bits wide) to the cache. However, at most, two simultaneous accesses to the same cache line are allowed. The latency of a read stall[2] is 61 cycles (@100 MHz), while a write stall lasts 59 cycles, excluding waiting times due to DDR2 refresh cycles and accesses performed by the Linux OS.

[1] 100 MHz, but transferring data on both the positive and the negative clock edge.
[2] The time needed to load a cache line from the main memory.

## 9.2. Dynamic CTs vs. Static CTs

Figs. 9.2 and 9.3 show COMRADE synthesis and simulation results of seven example kernels, most of which were taken from benchmarking suites such as Honeywell Stressmark [Hone97], MiBench [GREA01], MediaBench [LePM97] and CHStone [HTHT09]. The source codes actually used are given in Appendix D. For synthesis, we have employed Synopsys Synplify Premier DP (version E-2010.09) and Xilinx ISE 10.1.

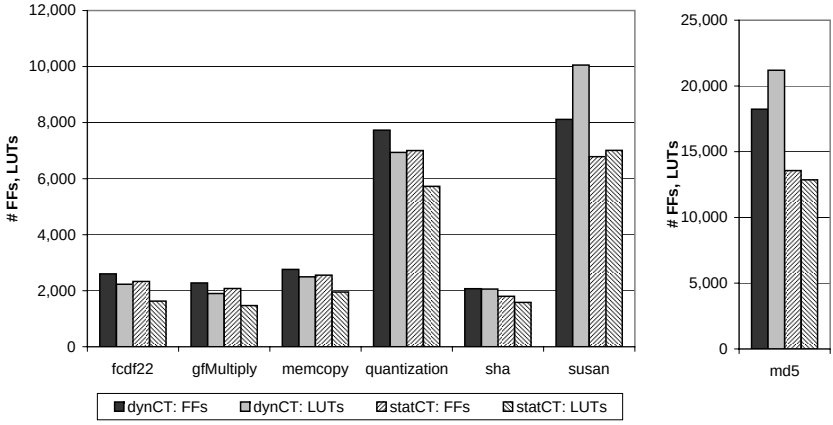


Figure 9.2.: Area comparison (FFs and LUTs) of dynamic vs. static CTs, synthesized for target frequency 100 MHz.

Regarding the FPGA area (Fig. 9.2), static CTs save about 26 % of the Virtex-5 look-up tables (LUTs) and 13 % of the flip-flops (FFs), because the sequencer as well as the operators require less token handling logic.

Interestingly, the runtimes of static CTs match those of dynamic CTs in all kernels examined in Fig. 9.3.

However, it *is* possible to find a kernel which *does* show differences in runtime. An example is the imbalanced_paths kernel shown in Listing 9.1: A loop runs through six iterations, the first five iterations executing a simple addition, while the last iteration executes the else part, which contains a high-latency divider (34 cycles).

Table 9.1 shows the resulting runtimes per loop iteration and in total. The start of iteration $i$ is defined here as the cycle in which the s value resulting from the $i$th loop body data path computation is stored in the s-loop mux (shown in

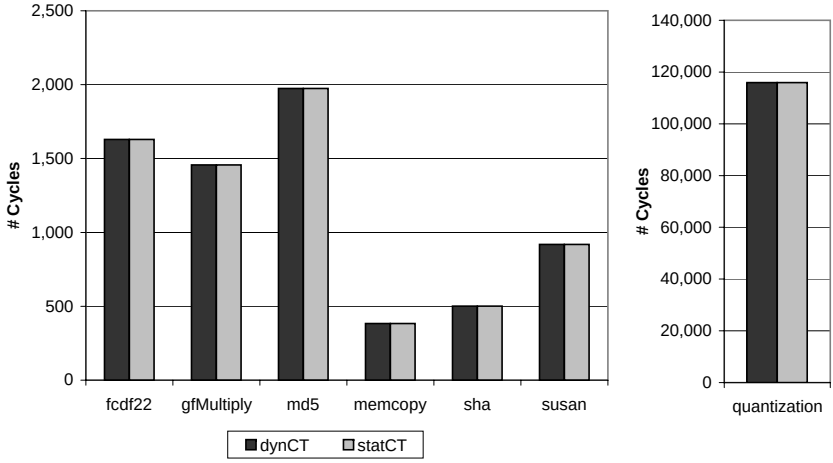


Figure 9.3.: Runtime comparison (number of simulation cycles) of dynamic vs. static CTs.

```
1  int main() {
2      unsigned int i, n = 6, s = 0;
3      unsigned int a, b, c, x = 0;
4
5      // HW starts
6      for (i = x; i < n; i++) {
7          if (i != 5) {
8              s += i;
9          } else {
10             a = i + 10;
11             b = a * 90 + 1 + i;
12             c = b / a;
13             s = s + 4 + c;
14         }
15     }
16     // HW ends
17     printf("s = %d\n", s);
18     return 0;
19 }
```

Listing 9.1: imbalanced_paths example.

| | Without Buffers | | With Buffer |
|---|---|---|---|
| Iteration | Dynamic CTs | Static CTs | Static CTs |
| init | 1 | 1 | 1 |
| 0 | 8 | 8 | 8 |
| 1 | 13 | 42 | 13 |
| 2 | 18 | 47 | 18 |
| 3 | 23 | 52 | 23 |
| 4 | 28 | 85 | 28 |
| 5 | 70 | 98 | 70 |
| Total | 74 | 102 | 74 |

Table 9.1.: Runtimes (in number of cycles) of the hardware generated from the code in Listing 9.1 for different parameter settings.

Fig. 9.4). The init row in Table 9.1 gives the cycle in which the loop mux accepts the initial s value. Columns two and three compare dynamic versus static CTs for the default case in which the operators are not equipped with output buffers (i.e., the node parameter QDEPTH is zero, cf. Section 7.1). Here, dynamic CTs perform noticeably faster than static CTs. A major difference occurs for iteration $i = 1$; Fig. 9.4 shows the performance bottleneck of the static CT version: Because the divider delays the execution of add2, the initial s value must remain in the output register of add1. Consequently, the resulting s value of iteration $i = 0$ stays in the output register of the loop mux, so that the new s value of iteration $i = 1$ cannot yet enter the loop mux.

If dynamic CTs are used (second column in Table 9.1), such a token jam does not occur: CTs actively move towards the stuck s values and extinguish them. This could be seen as the cancel tokens cleaning a congested drain, so that the divider latency affects only the last loop iteration.

Instead of cleaning the drain, another possibility would be to *elongate the drain tube*, by inserting buffers which can hold such superfluous, speculative s values. This could be done by extending add1 in Fig. 9.4 with a transparent output data buffer (parameter QDEPTH), large enough (48 entries are used here) to balance the latency of the divider path. The results in column four of Table 9.1 indeed show that the static CT model with buffer now performs as well as the dynamic CT model without buffers (column two). Inserting buffers in the dynamic CT model would not make sense, as the runtime would not improve further.

Table 9.2 compares the area resources (columns two and three) required by imbalanced_paths. The fully pipelined 32 bit divider alone requires about 3,200

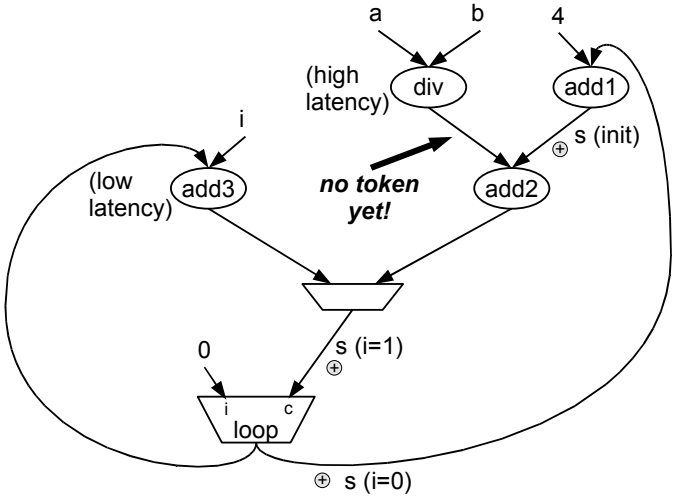


Figure 9.4.: Static CTs: token jam.

FFs and 4,200 LUTs; thus, columns four and five, showing the resource requirements without the divider area, better highlight the impact of parameter variations. According to this measure, the static CT version without buffers saves about 9 % of the FFs and 23 % of the LUTs versus static CTs due to the simplified operator and sequencer. When we add a buffer (as described above) to the static CT version to achieve the dynamic CT performance, the area requirements are still smaller than that of dynamic CTs (8 % FFs, 16 % LUTs), cf. rows one and three in Table 9.2.

To summarize our results, static CTs achieve the same performance as dynamic

| | FFs total | LUTs total | FFs w/o divider | LUTs w/o divider |
|---|---|---|---|---|
| Dynamic CTs, without buffers | 4,347 | 5,264 | 1,163 | 1,042 |
| Static CTs, without buffers | 4,232 | 4,999 | 1,061 | 806 |
| Static CTs, with buffer | 4,246 | 5,070 | 1,075 | 877 |

Table 9.2.: Area requirements (FFs and LUTs) of the hardware generated from the code in Listing 9.1, for different parameter settings; divider resources omitted in columns four and five.

CTs in most cases (cf. Fig. 9.3), while saving area resources. If dynamic CTs would actually perform better, buffer insertion could boost static CT performance, completely alleviating the dynamic CT advantage, while still saving area resources compared to dynamic CTs. In the following Sections we will therefore concentrate on the static CT model.

## 9.3. Operation Chaining

By default, COMRADE 2.0 parametrizes the CMDFG nodes so that each hardware operator will have a register holding the result of the operation, i.e., the parameter DEPTH is set to one[3]. Depending on the operation, its parameters (e.g., bitwidth, signedness, or CT scheduling type) and the parameters of the target architecture, two or more successive operations might have critical paths which are short enough to pack these operations into a single clock cycle. This **operation chaining** is a very effective technique, because it decreases both the latency and saves register resources, while the target frequency is not affected. A practical requirement for this optimization is knowledge about the critical path of each operator – this information is available to the COMRADE 2.0 compile flow through the MDF framework (cf. Section 7.2).

To understand the chaining algorithm we require some additional definitions. The chaining of an operation not only depends on the critical path of the operation itself, but also on combinatorial predecessors or successors that are already chained. Therefore, before chaining a node $n$, we consider its incoming and outgoing chained node paths.

**Definition 9.3.1.** Given a CMDFG and a distinct node $n$.

An **incoming chained node path of** $n$ is a (possibly empty) node path $icnp = (n_0, \ldots, n_k)$, so that $n_k$ is a predecessor of $n$, and each $n_i$ is unregistered. $ICNP(n)$ is the set of all incoming chained node paths of $n$.

An **outgoing chained node path of** $n$ is a non-empty node path $(n_0, \ldots, n_m)$, so that $n_0$ is a successor of $n$, each $n_i$ for $i < m$ is unregistered, and $n_m$ is registered. $OCNP(n)$ is the set of all outgoing chained node paths of $n$.

If $n$ would be chained, the operations in the incoming and outgoing chained node paths would take place in the same cycle as $n$. Note that an incoming chained node path does not have a register at its end, while an outgoing path

[3]Constant shifts and token forwarder nodes are not registered, because they collapse to pure wiring in hardware.

*has*. This register represents the end of the combinatorial node path – although being registered, this end node contributes to the critical path delay and is thus included in the outgoing chained node path definition.

For such node paths, we define the critical path delay.

**Definition 9.3.2.** Given a CMDFG and a node path $np = (n_0, \ldots, n_k)$.

The **critical path delay of *np***, $cpd(np)$, is the sum of the critical path delays of the nodes contained in $np$.

For operation chaining, only the longest combinatorial input and the longest combinatorial output are relevant; this is addressed by the next definition.

**Definition 9.3.3.** Given a CMDFG with a distinct node $n$.

The set of **maximal incoming chained node paths of** $n$, $MICNP(n)$, contains all incoming chained node paths of $n$ which have a maximal critical path delay, i.e.,

$$MICNP(n) := \{m \in ICNP(n) | \neg \exists k \in ICNP(n) : cpd(k) > cpd(m)\}.$$

Analogously, the set of **maximal outgoing chained node paths of** $n$, $MOCNP(n)$, contains all outgoing chained node paths of $n$ which have a maximal critical path delay, i.e.,

$$MOCNP(n) := \{m \in OCNP(n) | \neg \exists k \in OCNP(n) : cpd(k) > cpd(m)\}.$$

Note that $MICNP(n)$ is non-empty: If $n$ does not have an unregistered predecessor, $MICNP(n)$ contains exactly the empty set. $MOCNP(n)$ is non-empty, if there is a node path $(n, \ldots, m)$ along data edges, with $m$ being registered. In that case, $n$ is said to be **terminated**.

Finally, the critical path delays of these maximal chained node paths determine if $n$ may be chained.

**Definition 9.3.4.** Given a CMDFG with a terminated node $n$.

The **incoming critical path delay of** $n$, $ICritPD(n)$, is the critical path delay of a maximal incoming chained node path of $n$, i.e., for any node $m \in MICNP(n)$, $ICritPD(n) := cpd(m)$.

Analogously, the **outgoing critical path delay of** $n$, $OCritPD(n)$, is the critical path delay of a maximal outgoing chained node path of $n$, i.e., for any node $m \in MOCNP(n)$, $OCritPD(n) := cpd(m)$.

*ICritPD* is well defined, because all node paths in *MICNP* have the same critical path delay. This is analogously true for *OCritPD*.

We now present the algorithm used to chain the operators in a loop body. COMRADE 2.0 first generates a working set of chaining candidate nodes. These are registered CMDFG nodes having only incident data edges. Then, nodes are successively picked from the working set in data dependence order. For such a node $n$, we consider the incoming critical path delay and the outgoing critical path delay. The node $n$ is set to unregistered, if the sum of $ICritPD(n)$, $OCritPD(n)$, and the critical path delay of $n$ is smaller than the target cycle length. In the example in Fig. 9.5, $ICritPD(a) = 0$ and $OCritPD(a) = 3.60\,ns$, so the sum $ICritPD(a) + OCritPD(a) + 3.58\,ns = 7.18\,ns$ is smaller than the target cycle length of $10\,ns$. As a result, the register after node a is removed.

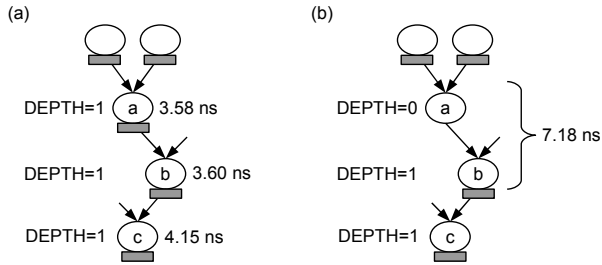


Figure 9.5.: (a) Before, (b) after chaining operation a; target cycle length: 10 ns (100 MHz).

Fig. 9.6 shows runtimes using the static CT model, comparing results for operation chaining switched on versus chaining switched off. Most of the examples show either no or only a marginal speed-up here, because they are dominated by memory accesses (which are not chained). The md5 kernel with its long pipelined data path achieves a speed-up of 1.34 in the chained version. This result includes the delay caused by cache stalls (which are not accelerated through operation chaining). Disregarding the stall delays, md5 would require 1,639 cycles (not chained) vs. 1,127 cycles (chained), making up a pure chaining speed-up of 1.45.

Fig. 9.7 shows the area savings achieved with operation chaining due to omitted registers. On average, 10 % of the FFs and 5 % of the LUTs are saved. The reasons for the latter are slight simplifications in the internal token logic of unregistered operations.

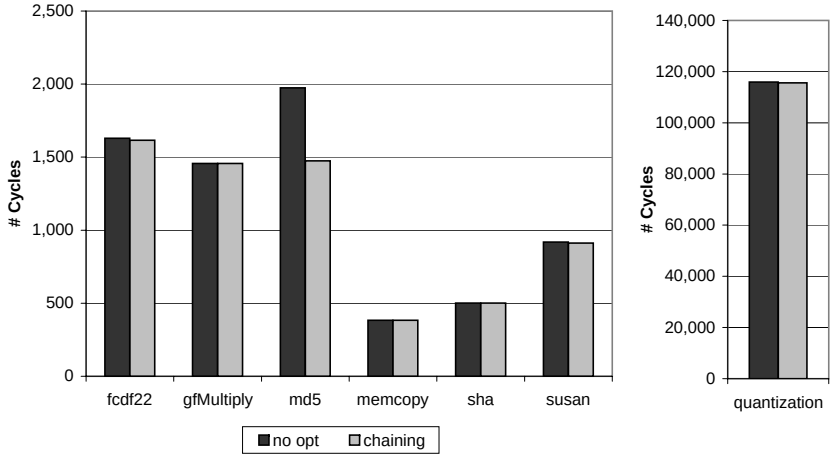


Figure 9.6.: Runtime comparison (number of simulation cycles), static CTs, operation chaining switched on vs. off.

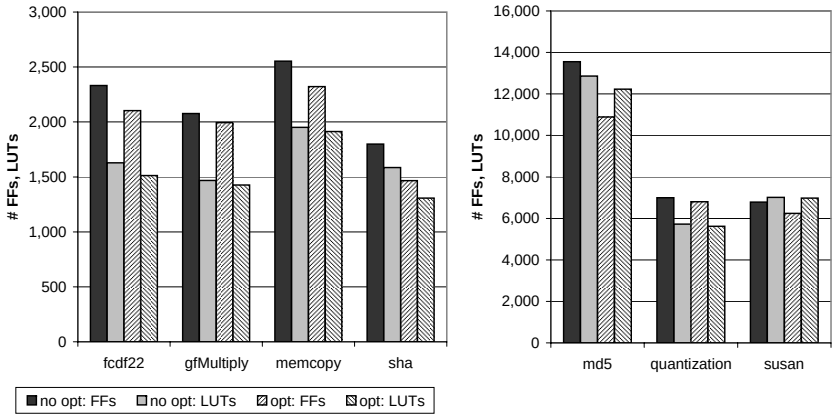


Figure 9.7.: Area comparison (FFs and LUTs), static CTs, operation chaining switched on vs. off; synthesized for target frequency 100 MHz.

## 9.4. Memory Access Reordering

By default, COMRADE 2.0 creates the CMDFG memory edges in program order. This can cause delays when the input data for a memory access are already available, but there is no token yet on the incoming *memory edge*. To better match the memory access (MA) order with the order induced by data dependences (and thus avoid such delays), COMRADE 2.0 reorders MAs by adjusting memory edges.

Fig. 9.8 shows an example: Assuming that it takes longer to compute A than to compute B ($t_A > t_B$), load2 will have to wait until load1 has finished, although the address for load2 is already available. In this case, it makes more sense to invert the memory edge and execute load2 *before* load1, irrespective of the program order.

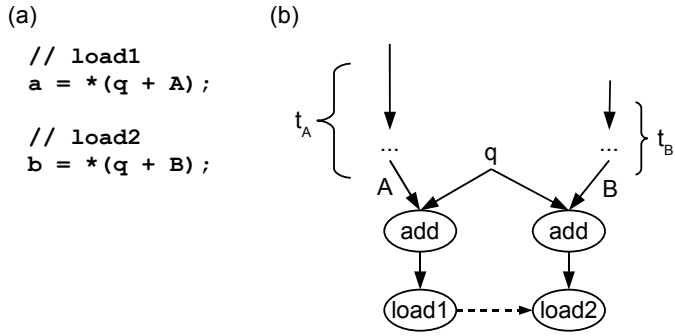


Figure 9.8.: (a) C code, (b) generated CMDFG section.

In general, altering the CMDFG memory edges while preserving the program semantics is only possible for independent memory accesses. There are numerous techniques for determining such independence information, ranging from simple rules (e.g., consecutive loads and pointers qualified by the restrict keyword are always independent) to complex analyses (e.g., pointer analysis [Stee96] [ShHo97] or loop-iteration space analysis [Pugh91] [Leng93]). However, this work concentrates on actual hardware generation principles rather than on such complex analysis techniques, which generally take place in the machine-independent parts of the compiler middle-end. Thus, we show here how to *use* independence information instead of how to compute it in a sophisticated manner. Therefore, we simply utilize the fact that loads are always independent, and we allow the user to manually indicate that stores are in-

dependent. Even with these simple techniques, we can measure noticeable performance gains (see below).

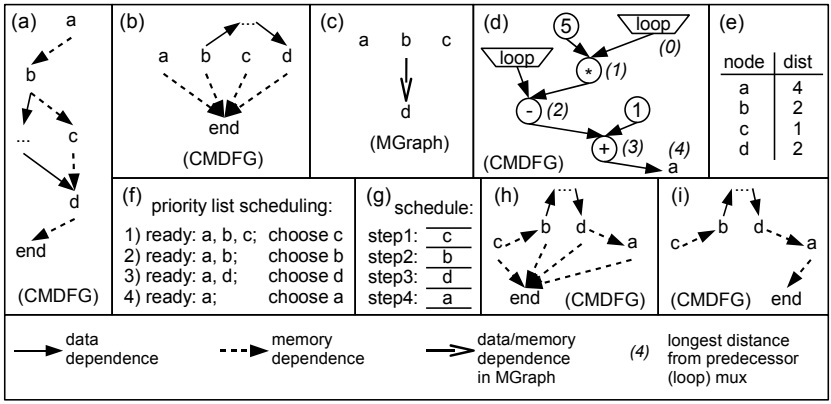


Figure 9.9.: The memory access reordering algorithm. (e) dist values for b, c, and d represent sample values (which are not deducible from (a)-(d)).

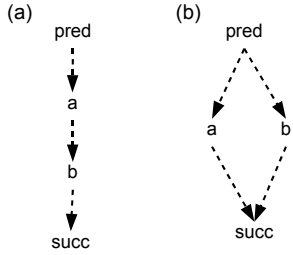


Figure 9.10.: (a) CMDFG section; (b) memory edge (a, b) removed, predecessors and successors adjusted.

Fig. 9.9 shows the algorithm used for reordering memory accesses (MAs). The algorithm is executed for each CFG node, thus reordering is applied on a per-CFG-node basis. Fig. (a) shows a CMDFG section of four MA nodes (a, b, c, d), which all refer to the same CFG node. Let these MAs be independent. Note that there is an (indirect) data dependence from b to d. For each memory edge connecting two independent MAs, we apply the transformation shown in Fig. 9.10, resulting in Fig. 9.9(b). We now build an auxiliary graph structure

MGraph (Fig. (c)), containing the MAs of the current CFG node as nodes and the CMDFG memory edges as edges, plus edges derived from CMDFG data dependences: Due to the indirect data dependence b → d, edge (b, d) is added to the MGraph. (No edges are inserted from memory dependences here, because a-d are independent and the end node is not an MA of the current CFG node.) Our goal is to match the MA order with the order induced by data dependences; because the input data is located in loop muxes at the beginning of a loop iteration, we have to find the longest data path from a loop mux to the MA. The path length is the sum of the operator latencies on the path, using expected latencies for variable latency operators. In Fig. (d) each operator has a latency of one cycle, resulting in the longest mux-to-MA path length of four for MA a. All these path lengths are stored in a distance table (e), from which a priority list of the MA nodes is built. Using this priority list, an MA schedule is generated through list scheduling [GWDL92]; this is shown in Fig. (f): The ready MAs are those MAs which are not yet scheduled, but their predecessors are already scheduled. First, all MAs but d are ready, because b is not yet scheduled. From the ready MAs, one with a smallest distance value is chosen and inserted into the schedule. In this manner, all nodes are added, resulting in the schedule in Fig. (g). According to the schedule, we insert re-serializing memory edges into the CMDFG (Fig. (h), note that a direct memory edge (b, d) is not inserted here due to the indirect data dependence between b and d), and finally remove the unnecessary memory edges. An edge to the end node is only required from the last MA (Fig. (i)).

MA reordering will of course only affect kernels for which the memory edge insertion in program order leads to avoidable delays. The results in Fig. 9.11 show that this applies to the susan kernel. Here, reordering saves 47 % of the runtime. For the other examples, the default program order sequence of the MAs already represents the optimal solution.

We omit a Figure showing area results here, because the resource requirements are not affected by MA reordering.

## 9.5. Memory Access Parallelization

So far, we have applied a very limited memory access model, using a single cache port to the MARC cache (cf. Fig. 9.1). However, MARC already supports multiple cache ports: Up to four simultaneous accesses to the cache are possible. In many cases the actual limit is two simultaneous accesses though, because the BRAM blocks underlying each cache line have just two access

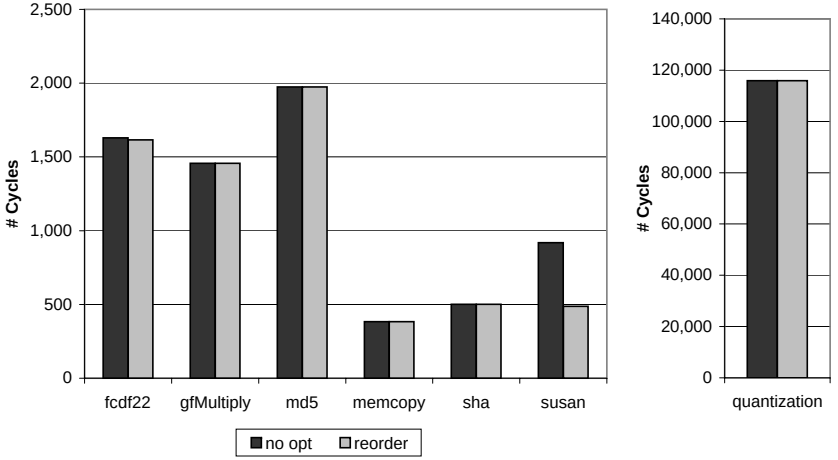


Figure 9.11.: Runtime comparison (number of simulation cycles), static CTs, memory access reordering switched on vs. off.

ports.

To support parallel cache accesses [GäSK08] in COMRADE 2.0, we reorganize the memory edges, and we map memory access (MA) nodes to actual MARC cache ports. For this we can re-use and slightly extend the MA reordering algorithm presented in the previous Section. To support $n$ parallel cache ports, we schedule independent MAs into $n$ columns in the Table in Fig. 9.9(g). MAs in column $i$ are then assigned cache port $i$, i.e., MAs in the same row may execute in parallel. New memory edges are thus inserted in a per-column top-down manner, resulting in $n$ parallel MA strings.

Fig. 9.12 shows runtime comparisons of one vs. two parallel cache ports; reordering is applied in both versions. The mean speed-up achieved is 1.15, with a minimum of 0.93 for gfmultiply, and a maximum of 1.42 for susan. The slow-down of gfmultiply is caused by an issue in the MARC memory back-end[4] and not related to our own research.

A general limitation of the overall speed-up here is related to cache stalls. The parallel cache ports, which enable the hardware kernel to access more than one cached data word in the same cycle, do not increase the bandwidth between the cache and the main memory. Thus, although multiple cache accesses can

[4] A fix for this is under way, but could not be integrated into our results anymore.

now be performed at the same time, the fraction of the execution time needed to fill and flush the cache lines is not affected by the MA parallelization. Due to Amdahl's Law, if $M\%$ of the execution time is improved by parallelization with $N$ cache ports, then the maximum speed-up is $\frac{1}{1-M\%+\frac{M\%}{N}}$. For $N = 2$, the maximum achievable speed-up thus ranges from 1.08 (gfMultiply) to 1.71 (md5) with an average of 1.47.

To examine the speed-up *disregarding* cache stalls, Fig. 9.13 shows runtime comparisons omitting the stall delays. The maximum speed-up achieved is 1.8 (1.33 on average), limited for two reasons: First, the overhead for starting a new loop iteration is not parallelized. For example, in the memcopy kernel, the time to execute the loop body is decreased from 16 cycles (one port) to 8 cycles (two ports), but two additional cycles are needed in both versions to start a new loop iteration, giving a speed-up of $\frac{18}{10} = 1.8$ instead of 2.0. Second, not all MAs are independent; e.g., out of five accesses per iteration in gfmultiply, only two are independent of each other.

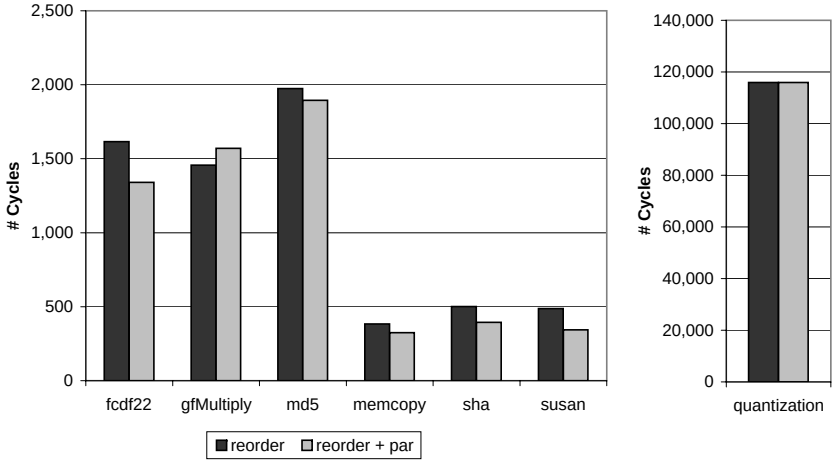


Figure 9.12.: Runtime comparison (number of simulation cycles), static CTs, reordering only vs. reordering with parallelization for two cache ports.

Fig. 9.14 compares the area requirements. Although the data path and sequencer require exactly the same resources for both one and two cache ports, there is a slight increase in the number of FFs in the two-port versions. This is due to the additional MARC cache port: The address and data buses from the

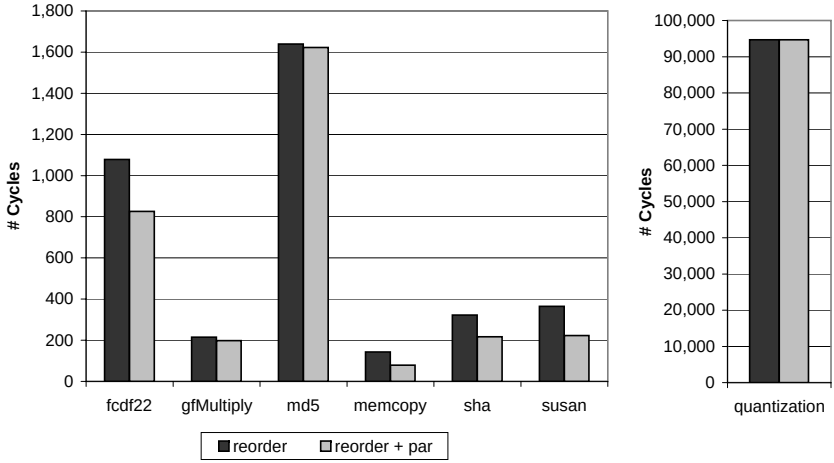


Figure 9.13.: Runtime comparison (number of simulation cycles), static CTs, reordering only vs. reordering with parallelization for two cache ports; cache stalls omitted.

hardware kernel to MARC are buffered in a hardware kernel wrapper to keep critical paths short. For two cache ports, twice as many of these buffers are required. Thus, this slight area increase is not caused by the hardware kernel itself.

## 9.6. Memory Localization

Using the optimizations presented so far (operation chaining, memory access (MA) reordering, and MA parallelization using two cache ports), we compare the runtimes of COMRADE-generated hardware kernels achieved on the ACE-M5 versus execution on the embedded PowerPC alone, as well as on an Intel Core2 CPU of a conventional PC. We derive the hardware kernel runtimes from the number of simulation cycles, considering the target frequency of 100 MHz. These numbers do not include delays due to CPU/RCU data transfer or RCU reconfiguration. We measure the CPU runtimes using a high performance counter on the Intel CPU and a cycle-accurate time base on the embedded PowerPC. For both CPU measurements the fastest available optimizations are

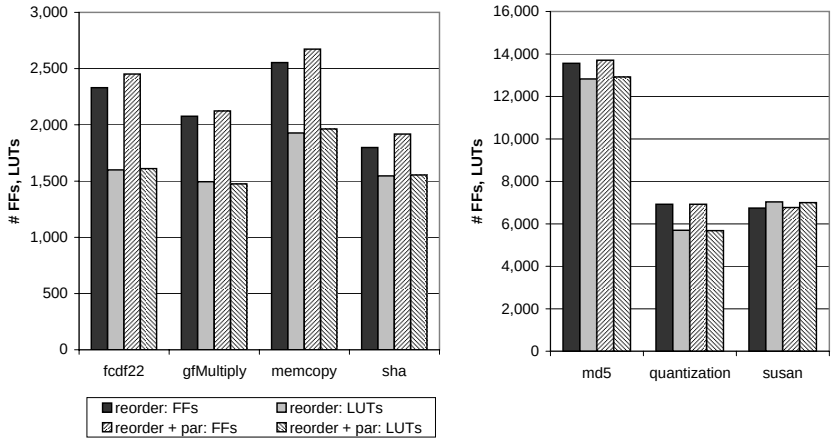


Figure 9.14.: Area comparison (FFs and LUTs): static CTs, reordering only vs. reordering with parallelization for two cache ports, synthesized for target frequency 100 MHz.

enabled[5].

Fig. 9.15 shows the results of these measurements. Three kernels (md5, memcopy, and susan) are actually executed faster on the 100 MHz RCU than on the embedded 400 MHz PowerPC. However, the mean hardware kernel speed-up is 0.84 versus the PowerPC, and down to 0.02 compared to the Intel CPU. This is a negative result especially when we notice that the power consumption of the embedded PowerPC (2.5 mW/MHz) [Ibmm10] lies roughly in the same range as the power consumption of a Xilinx FPGA-based RCU [GäKo04] [LSKH09].

An analysis of the hardware kernel bottlenecks reveals that the memory edges leading across loop conditions (cf. Section 5.3.2, Memory Accesses in Nested Loops) actually inhibit hardware pipelining of loop bodies, which is one of the most important techniques for hardware acceleration. To break this memory chain and actually pipeline loop bodies, two preconditions must be fulfilled. First, the MAs in the loop body have to be independent across loop iterations. Second, a considerably higher memory bandwidth is required, so that (at best) all MAs in the loop body can be executed in parallel. For this, a single cache is not appropriate, because it would have to arbitrate many accesses, increasing

[5]For the Intel CPU, we use the Microsoft Visual Studio compiler; for the PowerPC, we use a gcc cross-compiler.

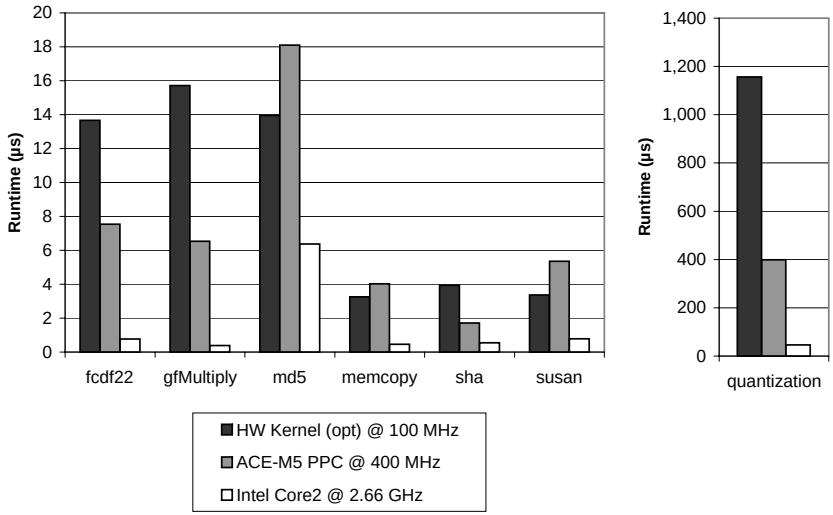


Figure 9.15.: Runtime comparison; HW kernels use operation chaining, MA reordering, and two parallel cache ports.

the latency of each single access. The cache currently integrated in MARC has the practical limit of four simultaneous cache accesses (being organized in two banks, each supporting two accesses at a time).

To be able to execute more MAs in parallel, we require more independent memories, which are (ideally) accessible without additional overhead. Possible approaches are memory localization techniques, offering many separate memories which can be located close to computational logic. Different local memory concepts include the many-cache model (used by the CHiMPS compiler [PBDM08]), MARC II [LaWK11], as well as local scratch pad memories. The many-cache model maps non-overlapping main memory sections to separate caches. If memory sections *do* overlap, cache coherency must be established through software. MARC II also relies on multiple caches, but keeps the caches coherent using hardware techniques. Furthermore, it supports speculative memory accesses. Scratch pad memories do not have a binding to the main memory per se; they are the ideal choice for the storage of temporary, local data which does not have to be written back to the main memory during a hardware-to-software transition. However, scratch pad memories are very broadly applicable and can even be used as caches (loading and flushing data is then implemented manually).

As MARC II has only been released very recently, it cannot be considered in greater detail in this work. Instead, we use local scratch pad memories to provide a higher memory bandwidth. This new feature, called **LMEM**, is shown in Fig. 9.16[6]. A hardware kernel can now access a configurable number of local memories in addition to the MARC cache ports. Each local memory (implemented in BRAMs on Xilinx Virtex family FPGAs) has a read and a write port which can be accessed independently. When the hardware kernel is not active, the CPU can access the local memories via memory mapping directly or (for better performance) program a **Local Paging Unit (LPU)**, which can copy data between the main memory and the local memories using a fast stream port (provided by MARC). Currently, a basic streaming version with a limited peak performance of 400 MB/s is used, while up to 3.2 GB/s are actually possible on the ACE-M5 (such an upgrade is under way). Memory access delays for the RCU can occur due to DDR2-SDRAM refresh cycles and OS access to the main memory.

COMRADE 2.0 supports LMEM in a semi-automated way that still requires some manual intervention. It is the programmer's choice to assign an array or pointer variable to either cache or a local memory. To indicate local memory, the suffix `_local` has to be appended to the pointer identifier. Each of these pointers may be used at most twice in the C program (once as a load, and once as a store), because each local memory has only two access ports. The programmer further has to take care of copying data to and from the local memories (e.g., by programming the LPU). The size and number of scratch pads, as well as possible parallel accesses, are set by compiler definitions. COMRADE then omits memory dependences completely, because all MAs are independent. Thus, the bottleneck-inducing memory chains have been broken.

To get an impression of the performance achievable with LMEM, we consider a localized version of the `md5` kernel. This kernel computes the MD5 checksum (128 bits) of an input message of arbitrary length. `md5` exhibits ideal properties for being accelerated through localization: It has a long loop body (having a latency of 192 cycles in our implementation), the loop body is pipelineable (we pipeline over a number of different messages here, using C-slow execution [LeRS83]), and all occuring memory accesses are independent.

However, an `md5` kernel compiled in a straightforward fashion does not yet work in a fully pipelined manner due to token jams. This is shown in Fig. 9.17(a); the numbers of the tokens denote the loop iteration the token is assigned to: After the left AT (1) has passed `op1`, it takes $n$ cycles (in the extreme

[6] For future work, the combination of LMEM and MARC II appears to be quite promising.

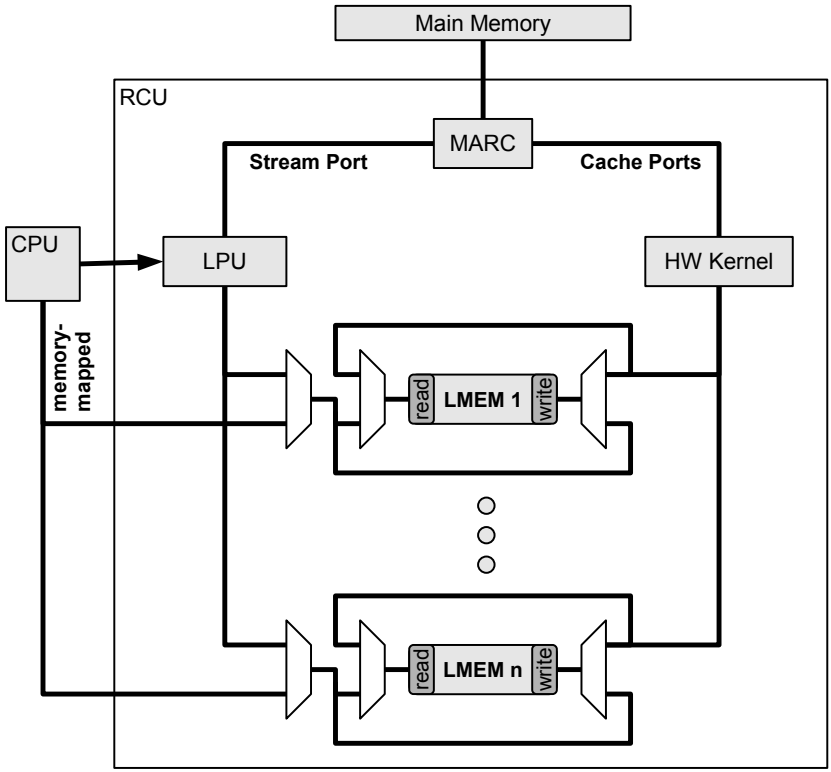


Figure 9.16.: The LMEM memory system.

case, $n = 192$ here) until op2 is started; in the meantime, the right AT (1) blocks the load node. In Fig. 9.17(b), a nop node extended with an output buffer of size $n$ holds the tokens which accumulate before op2, so that the load node can continue to feed input data into the long-latency data path, thus actually enabling pipelining. This technique can be understood as pipeline balancing in a dynamic context.

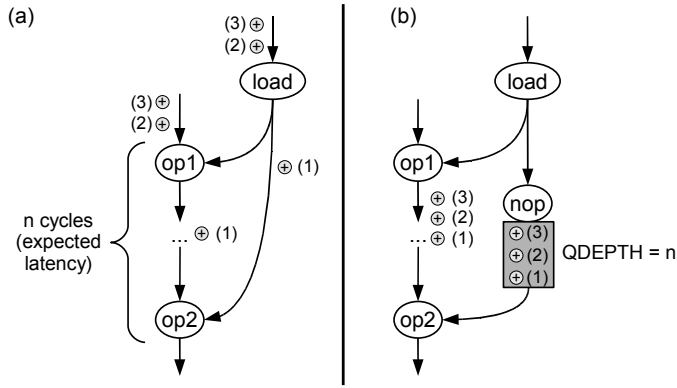


Figure 9.17.: md5 COCOMA section, (a) without output buffer (QDEPTH=0), (b) with output buffer (QDEPTH=n); tokens numbered according to loop iterations.

The optimal buffer size depends on the maximum number of loop iterations (i.e., the number of hash values to be computed, as we pipeline over a set of input messages here): For $n$ iterations, a buffer size of $n$ suffices. Once $n$ has reached 192 (the pipeline latency), it does not need to be increased any further (even for more than 192 hash values), because after that initial latency, tokens enter and exit a buffer at the same rate.

Because the localized md5 kernel has been studied during the final development phase of this work, buffer insertion for pipeline balancing has not been integrated into the general-purpose compile flow – instead, some heuristics are used for the md5 example, so that we can at least examine the effects on runtime and area requirements of this kernel.

Fig. 9.18 shows the measured runtimes using the static CT model (for dynamic CTs we got virtually the same results) for the buffer sizes 0, 32, and 192. Note the logarithmic scale: 1024 messages are processed 31x faster (due to pipelining) with a QDEPTH value of 192 compared to a design without buffers. The

pure HW kernel throughput of 1.95 GByte/s (@ 100 MHz), obtained with a buffer size of 192, can actually be achieved on the ACE-M5, which offers a throughput of 3.2 GByte/s between the main memory and the local memories.

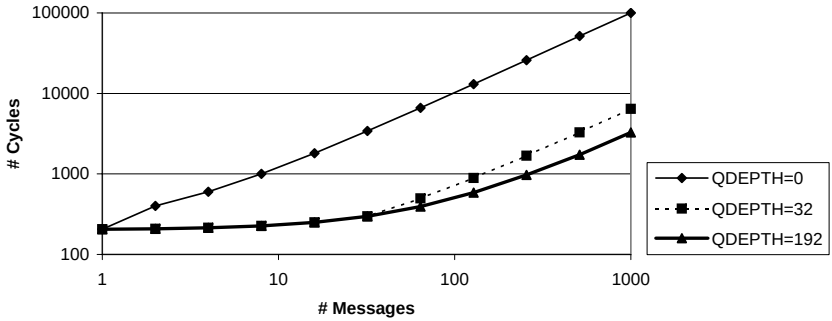


Figure 9.18.: md5 kernel, static CTs: number of simulation cycles over number of messages for different buffer sizes.

The resource requirements are shown in Fig. 9.19. For static as well as dynamic CTs the number of required LUTs increases with the QDEPTH buffer size, because the buffers are mapped to distributed memory (using LUTs) in the Virtex-5 FPGA. Static CTs here save 22 % of the FFs and 29 % of the LUTs compared to dynamic CTs. Unfortunately, only the static CT and dynamic CT md5 versions without buffers (QDEPTH=0) fit into the FPGA (XC5VFX70T) of our ACE-M5 test platform – we have actually run the static CT version on the board at 100 MHz. For the buffered versions an upgrade of the target FPGA would be necessary. For example, the XC5VFX200T chip with nearly three times as many FF and LUT resources would suffice for the md5 versions up to QDEPTH=128. In addition to the total number of FFs and LUTs available, we also need to take into account the number of LUTs that are actually usable as distributed memory. Although the XC5VFX200T chip offers 122,880 LUTs in total, only 36,480 LUTs are usable as distributed memory. Thus the QDEPTH=192 versions, requiring more than 44,000 of such LUTs (Fig. 9.20), do not fit into the XC5VFX200T. However, with newer FPGA generations, the amount of available logic resources increases (Virtex-6: up to 470k LUTs [Xili10]; Virtex-7: up to 1.2 million LUTs [Xils10]). FPGA area will no longer be the main performance-limiting resource. That role is likely to be taken up by external memory bandwidth in the future.

Assuming the availability of sufficient reconfigurable area, the HW kernel generated by COMRADE 2.0 with a buffer size of 192 would achieve a speed-up

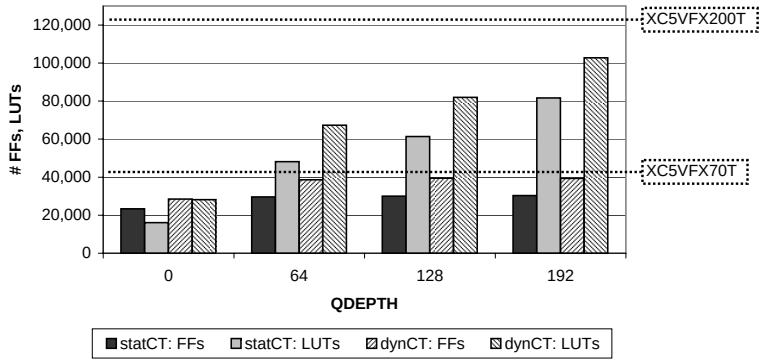


Figure 9.19.: md5 kernel: Virtex-5 area requirements for static and dynamic CTs; dotted lines indicate resources available on target FPGA.

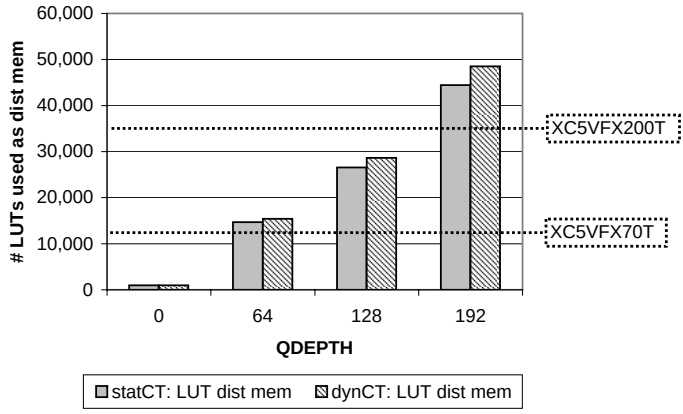


Figure 9.20.: md5 kernel: Virtex-5 distributed memory LUT requirements for static and dynamic CTs; dotted lines indicate resources available on target FPGA.

of 7x versus an Intel Core2 CPU @ 2.66 GHz, and a speed-up of 37x versus execution on the embedded superscalar PowerPC 440 @ 400 MHz of the ACE-M5 alone, as shown in Fig. 9.21. This illustrates the extreme performance potential of memory localization, improving the HW kernel speed-up by an order of magnitude.

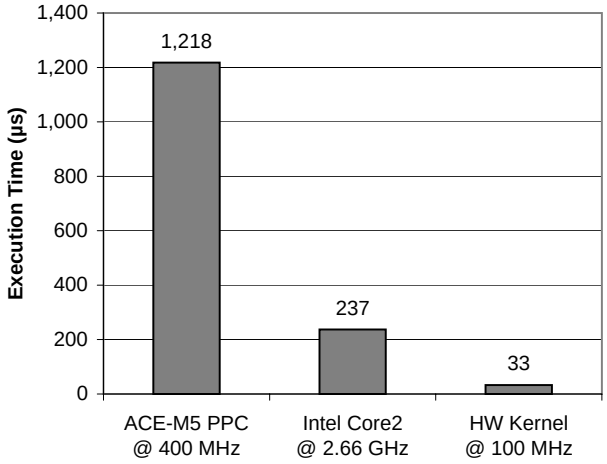


Figure 9.21.: md5 kernel: runtime comparison for 1024 messages, 512 bits each; HW kernel: QDEPTH=192.

# 10. Application-Level Study

Having analyzed isolated hardware kernels so far, we will now examine the speed-up on the application-level, i.e., the speed-up achieved for the execution of a complete application on an adaptive computer, where only parts of the application are actually accelerated by the RCU. As example application we use the wavelet image compression program (Versatility Stressmark) provided by the Honeywell Benchmark Suite [Hone97]. It compresses a 512x512 pixel gray scale image (8 bits per pixel) by successively applying the four steps (b)-(e) shown in Fig. 10.1. After the wavelet transform step (b), only one fourth of the resulting image (image areas A..G in the upper left corner) is processed by quantization (c), run-length encoding (d), and entropy encoding (e), finally resulting in a compressed bitstream (f).

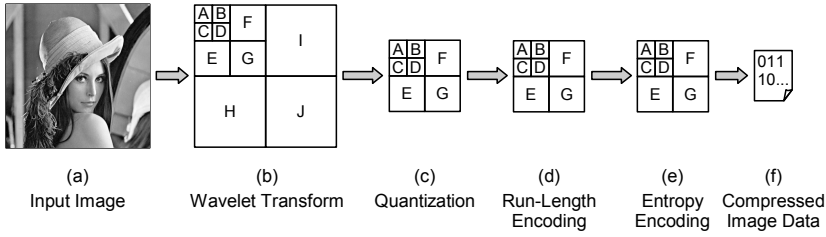


Figure 10.1.: Wavelet image compression steps.

The wavelet transform step is explained in more detail in Fig. 10.2. In fact the input image is transformed six times ((b)-(g) in the Figure), each time performing a (2,2)-biorthogonal Cohen-Deaubechies-Feauveau transform implemented in the lifting scheme [UyRB99]. Each transform stage produces a low-pass and a high-pass-filtered image section[1]. (b) transforms each row of the image, creating a low-pass section on the left and a high-pass section on the right hand side. (c) applies the same procedure to each image column, storing

[1] While the low-pass section is a scaled version of the original image, the high-pass section contains wavelet coefficients which are required to reproduce the original image from the scaled version.

the low-pass values at the top and the high-pass coefficients at the bottom of the image. This scheme of image row and column transformation is then reapplied to the top left quarter of the image ((d) and (e) in Fig. 10.2) and afterwards to the top left quarter of that quarter ((f), (g)).

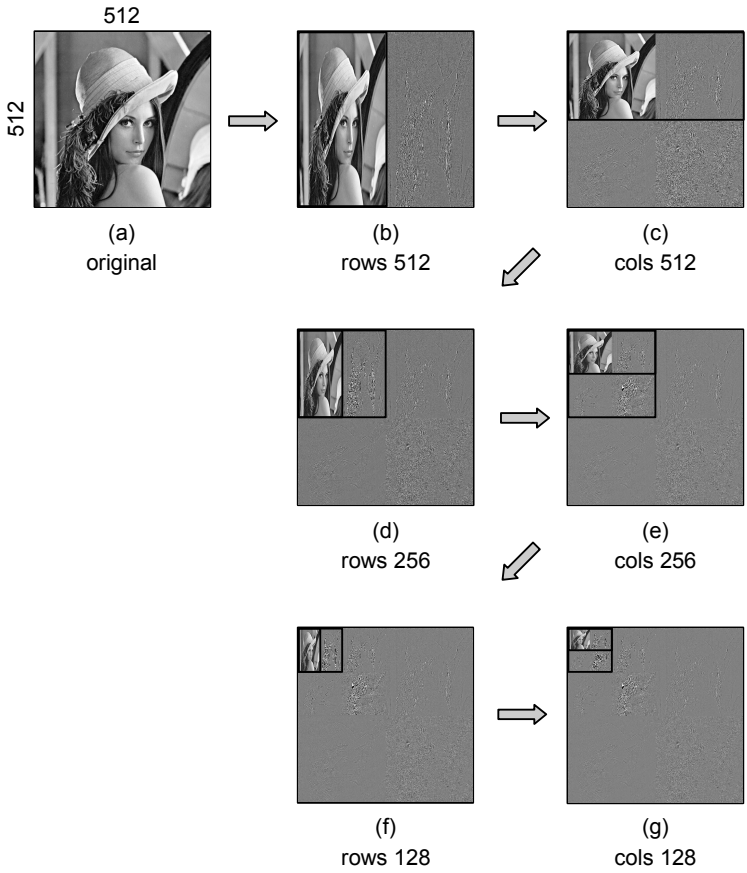


Figure 10.2.: Wavelet transform stages.

We have compiled[2] the compression algorithm and have executed it on the PowerPC 440 (PPC440) of the ACE-M5, using the popular Lena image (contained in the Honeywell Benchmark Suite [Hone97]) as input. Table 10.1

[2]Pure SW compilation using a GCC cross compiler, version 4.2.4; we have used the highest optimization effort level -O3.

| | Original | | Optimized (sw-opt) | |
|---|---|---|---|---|
| Wavelet Transform | 204,791 µs | 54.7 % | 204,809 µs | 94.0 % |
| Quantization | 156,751 µs | 41.8 % | 7,080 µs | 3.3 % |
| Run-Length Encoding | 9,293 µs | 2.5 % | 2,570 µs | 1.2 % |
| Entropy Encoding | 3,821 µs | 1.0 % | 3,362 µs | 1.5 % |
| Total | 374,656 µs | 100.0 % | 217,821 µs | 100.0 % |

Table 10.1.: Execution time of the wavelet compression program executed on the ACE-M5 CPU, original (columns two, three) and optimized version sw-opt (columns four, five).

(columns two and three) shows the runtime of each compression step, omitting file system delays (i.e., delays due to transfer between hard disk and main memory). We have discovered that this original implementation contains unnecessary computations. The quantization and run-length encoding steps transform all 10 image areas A..J, even though only areas A..G are actually processed by the final entropy encoding step and then stored in the output file. Furthermore, the quantization step iterates over image columns instead of rows, decreasing the cache efficiency. Just by fixing these two issues we have obtained an optimized version *sw-opt* of the program which saves more than 40 % of the runtime (columns four and five in Table 10.1).

Our goal is to create a hardware accelerated version of the optimized wavelet compression program. Obviously the wavelet transform step (94.0 % of the runtime) is the ideal hardware acceleration candidate. According to the results presented in Chapter 9, we apply memory localization to obtain a high memory throughput. An optimal solution would load the complete image into the local memory, so that the six wavelet transform stages can directly access the image data without interruption. However, the ACE-M5 LMEM cannot hold the complete image. As each pixel requires four bytes[3], the image requires 1,024 KB, while the Virtex-5 of the ACE-M5 offers a maximum of 666 KB of BRAM storage. Therefore we have to process the image section by section. For LMEM we choose 16 BRAM blocks with 4,096 bytes per block (the same configuration used for the `md5` kernel in the previous Section), making up a total LMEM size of 64 KB, i.e., each image section is 16,384 pixels in size.

We have accordingly adjusted the C algorithm[4] to operate on 16 arrays (subsequently referred to as *local arrays*), which model the BRAMs later used by the

[3]The C program uses the `int` data type to hold the gray value of a pixel, because the width of the data (originally eight bits) expands during processing.

[4]The resulting C code is shown in Appendix D.8.

| | Adjusted for COMRADE 2.0 (hw-opt) | |
|---|---|---|
| Wavelet Transform | 332,667 µs | 96.1 % |
| Quantization | 6,907 µs | 2.0 % |
| Run-Length Encoding | 2,541 µs | 0.7 % |
| Entropy Encoding | 4,177 µs | 1.2 % |
| Total | 346,292 µs | 100.0 % |

Table 10.2.: Execution time of the wavelet compression program executed on the ACE-M5 CPU alone, adjusted for compilation with COMRADE 2.0.

HW kernel[5]. These local arrays hold the image pixels initially, while holding low-pass values and high-pass coefficients during the operation of the algorithm. Before and after each actual wavelet transform, we copy the data from the original integer array to the local arrays and from the local arrays back to the integer array respectively. Table 10.2 shows the execution time measured for this altered program version, which we call *hw-opt*. Due to the copy operations, the total runtime has now increased by 59 % versus sw-opt in Table 10.1. The wavelet transform percentage has grown to 96.1 %.

We have compiled hw-opt (the source code is given in Appendix D.8) with COMRADE 2.0 and using static CTs. Setting `QDEPTH` to 16 suffices here, because the data path lengths are much shorter than for the `md5` kernel in the previous Section. Simulation results show that the compiled HW kernel requires 5,174 cycles to transform an image section of 16,384 pixels. In fact the HW kernel transforms 16 pixels every five cycles (due to loop-carried dependences of length five) making up a throughput of 3.2 pixels per cycle. Note that the HW kernel throughput is the same regardless of whether the pixels have been copied from image rows or image columns; instead, the column access delay influences the data transfer time between main memory and LMEM.

When copying data between main memory and LMEM, note that only for the first wavelet stage (`rows 512`) the LPU is able to transfer a whole image section by a single burst transmission. For the remaining row stages (`rows 256`, `rows 128`) each row has to be transferred by a separate burst, because the memory footprints of the rows are not directly contiguous in those cases. Thus the LPU must be programmed multiple times. For the column-wise transforms, even the `cols 512` stage requires a separate burst per column, again because the columns are not stored in directly adjacent memory regions. Furthermore, column pro-

[5] As memory localization has only been integrated in the final development phase of this work, this transformation is currently done manually; cf. Section 9.6.

cessing requires non-unit strided main memory accesses, because 511 pixels must be skipped in order to access the next pixel of a column. Unfortunately non-unit strides are not implemented in the stable memory back-end for the ML507 (MARC). Professor Koch's group is currently extending MARC II to provide this capability and also allow wider data transfers of 256 bits per cycle.

For this thesis, we were able to test a preliminary version of this fast streaming implementation which already works correctly in post-synthesis simulations, but not yet in real hardware (most likely due to vendor tool bugs in timing optimization). Combining this MARC II prototype with the COMRADE-generated HW kernel and LMEM, we have successfully synthesized a design targeting the ML507, which works correctly up to the post-synthesis simulation. We call this design *wave stream*.

In order to show the correct behavior of the HW kernel even after place-and-route, we have built the streaming-less design *wave mmap* which uses memory mapping instead of LPU-based streaming to transfer data between main memory and LMEM. For this design, even the post-place-and-route simulation has been successful, showing that the HW kernel works correctly at the designated ACE-M5 frequency of 100 MHz.

In the following we consider the results obtained for wave stream. Fig. 10.3 shows the area required by the complete FPGA design. About half of the FFs, LUTs, and BRAM blocks are used.

Table 10.3 and Fig. 10.4 show the FPGA resource distribution for the design components. The complete hardware design consists of the HW kernel (data path and sequencer), LMEM (LPU and local memory), buffers and muxes (which connect HW kernel, LMEM, and MARC II), a MARC II prototype (containing a techmod[6] for the connection to the MCI bus, and a stream port to buffer streamed data), and additional ML507 board support components (BSC, including on-chip buses, DDR2 controller, and peripherals such as compact flash, UART, and ethernet). The biggest fraction of the FFs and LUTs are required by the data path and the BSC; BRAM is used mainly by LMEM, MARC II, and the BSC.

Fig. 10.5 shows the (wave stream) runtimes for each wavelet stage obtained by post-synthesis simulation. Obviously the column stages run much longer than the row stages. This is due to the strided main memory accesses which decrease the bandwidth to 4 (instead of 32) bytes per cycle. Altogether we have measured 2,787,187 simulation cycles, which translates to 27,872 µs on the ACE-M5. This runtime includes HW kernel computation, data transfers

[6] A techmod is an adapter which connects a certain bus technology to MARC II.

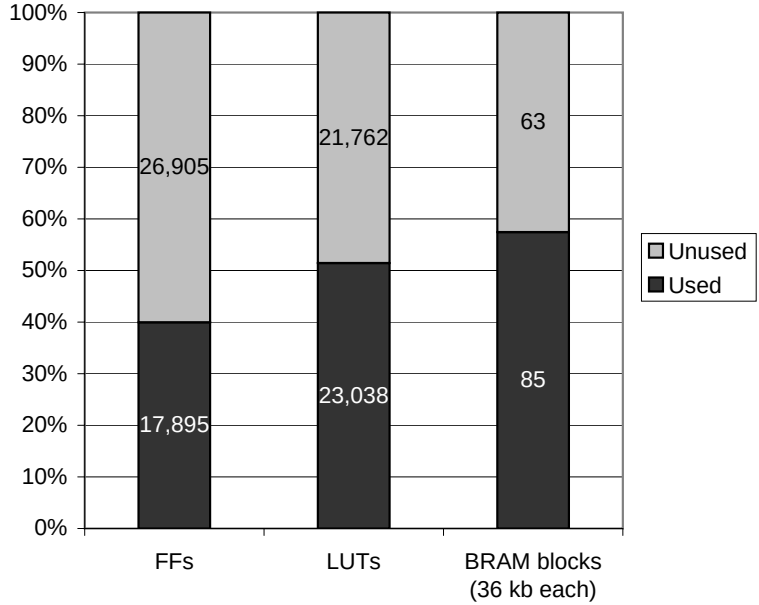


Figure 10.3.: Wave stream: area requirements for the ACE-M5 RCU.

| | FFs | LUTs | BRAM blocks (36 kb each) |
|---|---|---|---|
| Board support components | 7,458 | 9,281 | 39 |
| MARC II stream_port_256 | 481 | 401 | 4 |
| MARC II techmod_mci | 1,413 | 793 | 25 |
| buffers + muxes | 1,721 | 11 | 0 |
| local memory | 907 | 2,484 | 17 |
| LPU | 577 | 725 | 0 |
| HW kernel: sequencer | 267 | 874 | 0 |
| HW kernel: data path | 5,071 | 8,469 | 0 |

Table 10.3.: Wave stream: area distribution.

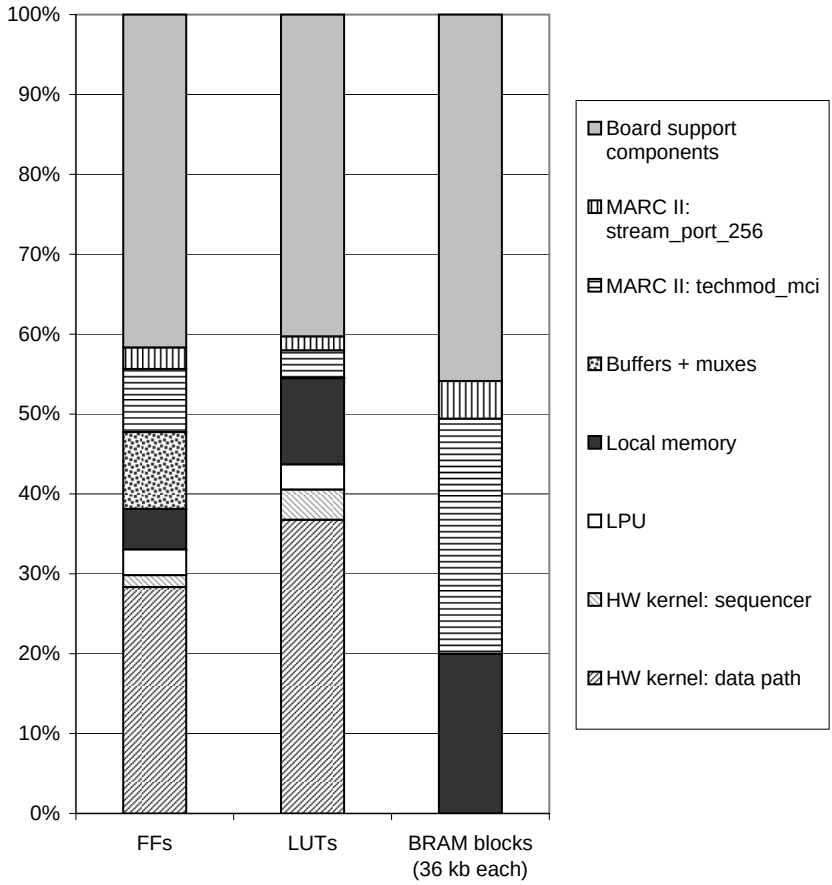


Figure 10.4.: Wave stream: area distribution.

between DDR2-SDRAM and LMEM (even accounting for DRAM refresh cycles), as well as memory-mapped accesses[7] to the HW kernel and the LPU. During simulation, the LPU has been used 2,592 times, while the HW kernel has been started 42 times. Compared to the fastest software-only wavelet transform version run on the PPC440, the HW kernel achieves a speed-up of 7.3x.

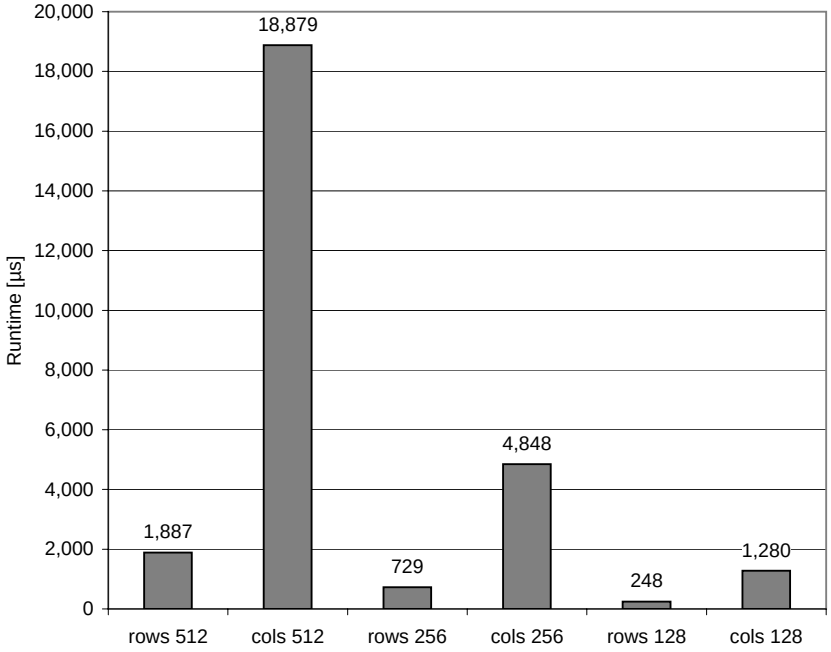


Figure 10.5.: Wave stream: ACE-M5 runtimes per wavelet stage, obtained by post-synthesis simulation; total application runtime: 27,872 µs.

Taking also the quantization, run-length encoding and entropy encoding steps into account (Tab. 10.4), we obtain a total ACE-M5 runtime of 41,497 µs. Fig. 10.6 compares this result to software-only execution on the ACE-M5 CPU (PPC440) and on an Intel Core2 CPU. The COMRADE 2.0-based wave stream application achieves a considerable speed-up of 5.2x versus the fastest PPC440 version (sw-opt), even though the PPC440 runs at 4x the clock frequency of the RCU. Compared to sw-opt run on the Core2, the ACE-M5 is only 10 times

[7] To transfer live variables to the HW kernel and to configure the LPU.

| | Processing Unit | ACE-M5 Runtime [µs] |
|---|---|---|
| Wavelet transform | RCU | 27,872 |
| Quantization | CPU | 6,907 |
| Run-length encoding | CPU | 2,541 |
| Entropy encoding | CPU | 4,177 |
| Total | | 41,497 |

Table 10.4.: Wave stream: total ACE-M5 runtime.

slower, although the Core2 frequency is 26 times higher.

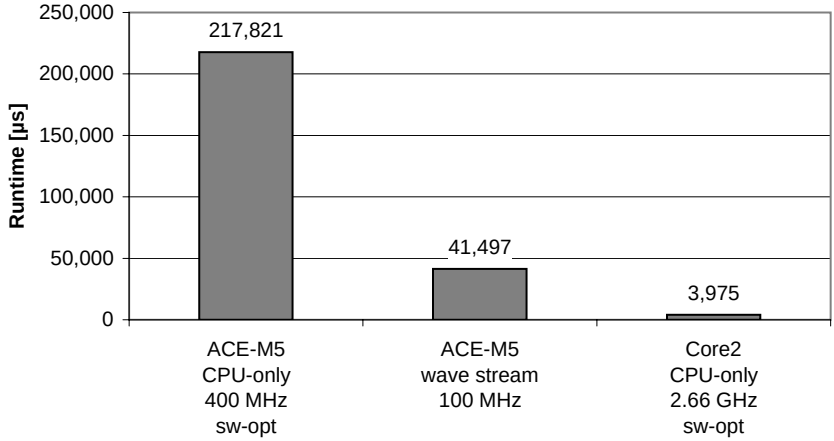


Figure 10.6.: Wavelet compression: execution time comparison, different compute platforms.

The comparison between the ACE-M5 and the Core2 is quite unfair, though. The ACE-M5 has to process the image step by step in an out-of core manner due to its limited internal BRAM memory[8]. Thus, the image data (and the wavelet coefficients, respectively) are copied multiple times between main memory and LMEM. The Core2 instead possesses a 4 MB level-2 cache, so that it can operate entirely on the cache.

This limitation will fall away with newer FPGA versions, which offer up to 4 MB (Virtex-6) and 8 MB (Virtex-7) of internal BRAM memory. Assuming

[8] 64 KB are actually used; 666 KB would be available. However, using more than 64 KB would not significantly decrease the memory transfer load between main memory and LMEM. For this, at least 1 MB would be required.

that the image is transferred from main memory to LMEM only once (before the wavelet transform stages) and the resulting wavelet coefficients are transferred back only once (after the last stage), Tab. 10.5 (column two) shows an estimation of the resulting runtime based on our wave stream measurements[9]. While the ACE-M5 wavelet transform runtime now nearly draws level with the Core2 (column four), the so far unaccelerated remaining steps quantization, run-length encoding, and entropy encoding (QRE) become a bottleneck. COMRADE 2.0 could be used to generate adequate HW kernels, so that these three steps can also be accelerated by the ACE-M5 RCU. Assuming the same speed-up for these kernels over the PPC440 software versions as estimated for the wavelet transform step, the ACE-M5 could achieve a slightly better runtime (column three) than the Core2. This result is very promising in terms of power consumption. While the ACE-M5 requires ca. 10 W (including CPU and RCU), the Core2 dissipates ca. 65 W [Inte11], so that the ACE-M5 would save about 87 % of the Core2 power.

And there is still more yet unexploited potential left. More internal memory together with more reconfigurable area resources would allow even better speed-ups with larger HW kernels (e.g., executing the wavelet transform for 32, 64, or more pixels in parallel instead of currently 16).

To summarize, this application-level study gives a first glance on the performance (in terms of speed-up, area requirements, and power) achievable on the ACE-M5 with COMRADE 2.0. While the currently achieved speed-up obtained over an embedded CPU is already significant, an FPGA with more internal BRAM memory (e.g., Virtex-6, Virtex-7) is required to better exploit the potentials offered by our LMEM architecture. For a more realistic rating of COMRADE 2.0-generated kernels as well as the ACE-M5, a larger number of benchmarks needs to be evaluated. Especially for examples with a higher degree of irregularity (e.g., if/else structures), we expect even higher HW kernel speed-ups due to a greater amount of pipeline stalls in the competing CPU.

[9] The wavelet transform step alone achieves a speed-up of 63x over the embedded PowerPC. We estimate the QRE accelerated runtime through dividing the PowerPC runtime by 63, thus estimating the wavelet transform speed-up for the QRE steps.

| | ACE-M5 Runtime [µs] (red. traff.) | ACE-M5 Runtime [µs] (red. traff.) (QRE accel.) | Core2 Runtime [µs] |
|---|---|---|---|
| SDRAM to LMEM | 527 | | |
| Wavelet: rows 512 | 828 | | |
| Wavelet: cols 512 | 828 | | |
| Wavelet: rows 256 | 207 | | |
| Wavelet: cols 256 | 207 | | |
| Wavelet: rows 128 | 52 | | |
| Wavelet: cols 128 | 52 | | |
| LMEM to SDRAM | 532 | | |
| Total Wavelet | 3,232 | 3,232 | 2,911 |
| Quantization | 6,907 | 109 | 513 |
| Run-length encoding | 2,541 | 40 | 126 |
| Entropy encoding | 4,177 | 66 | 425 |
| Total | 16,857 | 3,447 | 3,975 |

Table 10.5.: Wave stream: ACE-M5 runtime estimated for reduced memory traffic (column 2) and additional HW kernels for accelerating quantization, run-length encoding, and entropy encoding (QRE, column 3); comparison to the Core2 CPU (column 4).

# 11. Summary and Future Work

We have examined new approaches to high-level language compilation for adaptive computers, especially in the hardware back-end. While most of our techniques are language-independent, we have chosen to examine the compilation of ANSI C, which is still dominant in the embedded systems domain. An important premise of our approach is the broad language support including pointers and nested loops, instead of using just a subset of C to represent simple data flow in C syntax. To cope with the complex dependences and variable-latency operations arising from C constructs, we have investigated the promising dynamic scheduling approach.

As the hardware generated by our compiler is intended to be used as hardware accelerator, we predominantly optimize for runtime. To this end, we have employed a number of techniques: pipelining of data paths, even operator-internal pipelining, and speculative execution. We refine the latter by operation canceling, a technique which allows for canceling mis-speculated operations. To be able to model such techniques in the context of the complex dependences arising from C, we required a new micro-architectural model, because simple (control) data flow graphs did not suffice anymore. Therefore we have developed COCOMA, a low-level intermediate representation that fills the gap between the control flow graph (CFG) and the hardware description language (HDL) representation. This token-based model is sufficiently flexible to accommodate even dynamic systems with variable latencies.

We have explained in detail how a CFG can be mapped to a COCOMA instance. For this, normalizing the CFG into a structured form with top-testing loops has proven practical. From such a normalized CFG, a COCOMA instance is created in two steps. First, nodes and dependence edges are created; this produces a control memory data flow graph (CMDFG). Second, a dynamic scheduling sequencer is generated from the dependences encoded by the CMDFG graph structure. This sequencer acts as glue logic between the operators. CMDFG and sequencer together represent the COCOMA instance. Once this construction is complete, the compilation to the HDL is straightforward. The sequencer mainly produces combinatorial logic, while the CMDFG results in data paths composed from instantiated, word-wide hardware operators. For

this it has proven helpful to resort to parametrizable operators, which we provide in the Modlib operator library developed according to the requirements of this work. Simple extensibility and platform independence are important features of the library, significantly contributing to its usability.

We have shown the practicality of our approaches by developing the COMRADE 2.0 compiler framework based on COMRADE 1.0 [Kasp05]. Of the latter, we have re-used the front-end and adjusted it to meet the requirements of our new back-end. For example, the new front-end removes goto statements, so that the remaining compile passes work on a structured program representation. Our new back-end transforms the CFG via COCOMA to the HDL Verilog. We have tested the compiler with several practical input programs (synthetic regression tests as well as real-world examples) and obtained correct simulation results. This is a substantial progress over COMRADE 1.0, which implemented only a rudimentary back-end with far fewer features, and which produced correct Verilog output only for very simple C examples.

As far as we know, our work is the first study comparing simulation and synthesis results of dynamic versus static cancel tokens (CTs). Static CTs have turned out to be the better alternative, because they save area resources while achieving the same runtime in most cases. In the cases where dynamic CTs would actually achieve a better runtime, static CTs can be supported by adjusting parameters of the surrounding logic (e.g., increasing data and token buffer sizes) to again match the performance of dynamic CTs, but requiring less chip area.

We have developed and examined several optimizations at the COCOMA level. Operation chaining can save runtime, if the runtime is dominated by data path latencies. We have measured speed-ups of up to 25 %. Furthermore, chaining removes unnecessary registers and saves 5 to 10 % of the area resources. Speed-ups have also been measured for memory access reordering (up to 42 %) and memory access parallelization (up to 47 %), without adversely affecting the area requirements. But even combining these three techniques does (in most cases) not suffice to outperform an embedded CPU. The main reason for this is the limited memory bandwidth when using a central cache.

A considerable bandwidth increase is possible with memory localization. Therefore we have developed the LMEM architecture, which provides independent local memories, all of them accessible in parallel by the hardware kernel. A local paging unit allows for fast, stream-based data transfer between the main memory and LMEM. LMEM can be used very effectively if the memory allocated by the target application can be separated into several independent areas. An example hardware kernel running at 100 MHz computes MD5 hash

values 37x faster than an embedded CPU at 400 MHz, and still 7 times faster than a conventional Intel CPU at 2.66 GHz. For a wavelet image compression algorithm compiled with COMRADE 2.0 and using LMEM, we have measured an application-level speed-up of 5x over the embedded CPU, including data transfer delays between main memory and LMEM. The latter example runs only 10x faster on the conventional Intel CPU than on the adaptive computer, although the Intel CPU has a 26x higher clock frequency.

Being an academic project with limited manpower, we were not able to fix all of the remaining bugs in the system (primarily in the externally provided C front-end). However, these known bugs can be bypassed through slight adaptations of the input C code. An example is the missing support for the address operator (&). The compiler should provide this by first eliminating unnecessary uses and replace the remaining occurances by pointers. As a workaround, the user currently replaces address operators manually.

To find bugs in the COCOMA implementation, the formal definition in Appendix A has proven to be extremely helpful. The COMRADE 2.0 C++ code adheres very closely to these formal definitions, so that a bug found during hardware kernel simulation can be immediately identified and fixed.

A very practical approach considering the operator library was to embed operator-specific token logic inside the Modlib operators. This made the unified operator interface possible, which in turn simplified the design of the token flow and the implementation of the sequencer generation algorithms.

To enhance the design and debugging flow during our work, we have integrated mechanisms for graphical output of COCOMA instances as graph-like structures. In this manner, the static COCOMA structure can be visualized after compilation, while dynamic behavior animations can automatically be generated from the simulation output.

Despite the efforts and contributions of this work, we have encountered many venues for future work and research. Even though the original COMRADE 1.0 front-end as well as the underlaying SUIF2 framework contain several high-level optimization passes, far more known compiler optimizations are not yet taken advantage of. For this it seems to be reasonable to migrate to another underlaying framework such as LLVM [Latt02] or Scale [Scal11], offering existing implementations of useful optimizations. Several pointer analysis techniques available in these frameworks would be very helpful to automate memory reordering, memory parallelization, and memory localization.

Another suggestion to enhance the COMRADE 2.0 front-end is the integration of floating point support. This would be possible with little effort, because the

required hardware operators are already contained in Modlib, and hardly any change to COCOMA would be necessary.

In the back-end, speculative memory accesses would be an interesting extension. Here, one has to distinguish between control speculation and data speculation. The former executes a memory access when data dependences are fulfilled, without being known if the access actually has to be executed (i.e., there is no token yet at the incoming control edge). The latter executes the memory access even without fulfilled data dependences, e.g., using a predicted address value. Control speculation would require adaptations (such as an undo function for speculative writes) mainly in the memory back-end, while data speculation needs to adjust the COCOMA token flow as well.

The usability of COMRADE 2.0 would be greatly improved by better automation in the context of the very promising memory localization technique. This includes finding independent memory sections, re-arranging them, and programming the local paging unit accordingly.

Finally, the configuration scheduling technique [KaVK05], which is already existing, needs to be used by the compiler, so that the compile flow is actually completed down to the final FPGA configurations.

To conclude, this work has shown that dynamic scheduling is an effective and appropriate method for high-level language compilation for adaptive computers. Besides exploring interesting approaches and gathering novel results, we have discovered many further venues for future research.

# Bibliography


[ADHL00] Aigner, G. ; Diwan, A. ; Heine, D. L. ; Lam, M. S. ; Moore, D. L. ; Murphy, B. R. ; Sapuntzaki, C.: An Overview of the SUIF2 Compiler Infrastructure / Stanford University. 2000. – Technical Report

[BöJa66] Böhm, C. ; Jacopini, G.: Flow diagrams, turing machines and languages with only two formation rules. In: *Communications of the ACM* 9 (1966), no. 5, pp. 366–371

[BrGa03] Brej, C. ; Garside, J.: Early Output Logic using Anti-Tokens. In: *Proceedings of the International Workshop on Logic Synthesis (IWLS)*, 2003

[Budi03] Budiu, M.: *Spatial Computation*. Carnegie Mellon University, Pittsburgh, USA, School of Computer Science, Diss., Dec. 2003

[BuNa08] Buyukkurt, B. ; Najjar, W. A.: Compiler Generated Systolic Arrays for Wavefront Algorithm Acceleration on FPGAs. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 655–658

[CaHW00] Callahan, T. J. ; Hauser, J. R. ; Wawrzynek, J.: The Garp architecture and C compiler. In: *IEEE Computer* 33 (2000), Apr., no. 4, pp. 62–69

[CaMS01] Carloni, L. P. ; McMillan, K. L. ; Sangiovanni-Vincentelli, A. L.: Theory of latency-insensitive design. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20 (2001), Sep., no. 9, pp. 1059 –1076

[CFHJ06] Cong, J. ; Fan, Y. ; Han, G. ; Jiang, W. ; Zhang, Z.: Platform-Based Behavior-Level and System-Level Synthesis. In: *Proceedings IEEE International System-on-Chip Conference (SOC)*, 2006, pp. 199–202

[CFRW91] Cytron, R. ; Ferrante, J. ; Rosen, B. K. ; Wegman, M. N. ; Zadeck, F. K.: Efficiently computing static single assignment form and the control dependence graph. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13 (1991), Oct., no. 4, pp. 451–490

[Dijk72] Dijkstra, E. W.: *Chapter I: Notes on structured programming*. London, UK : Academic Press Ltd., 1972. – pp. 1–82.

[Drey94] Dreyer, S.: *Die Umwandlung hierarchischer Netzlisten vom Logik-Austauschformat SLIF in die Hardware-Beschreibungssprache Verilog*. Institut für Theoretische Informatik, Technische Universität Braunschweig, Germany, Jun. 1994

[ErHe94] Erosa, A. M. ; Hendren, L. J.: Taming Control Flow: A Structured Approach to Eliminating GOTO Statements. In: *Proceedings of the International Conference on Computer Languages (ICCL)*, 1994, pp. 229–240

[FeOW87] Ferrante, J. ; Ottenstein, K. J. ; Warren, J. D.: The Program Dependence Graph and Its Use in Optimization. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9 (1987), no. 3, pp. 319–349

[GäKo04] Gädke, H. ; Koch, A.: Wavelet-based Image Compression on the Reconfigurable Computer ACE-V. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)* vol. 3203/2004, Springer Berlin / Heidelberg, Aug. 2004, pp. 1006–1010

[GäKo08] Gädke, H. ; Koch, A.: Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens. In: *Proceedings of the 4th International Workshop on Reconfigurable Computing (ARC)* vol. 4943/2008. Berlin, Heidelberg : Springer, Aug. 2008, pp. 185–195

[GäSK08] Gädke, H. ; Stock, F. ; Koch, A.: Memory Access Parallelisation in High-Level Language Compilation for Reconfigurable Adaptive Computers. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 403–408

[GäTK10] Gädke-Lütjens, H. ; Thielmann, B. ; Koch, A.: A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2010, pp. 475–482

[GCHB05] Le Gal, B. ; Casseau, E. ; Huet, S. ; Bomel, P. ; Jego, C. ; Martin, E.: C-based Rapid Prototyping For Digital Signal Processing. In: *Proceedings of the 13th European Signal Processing Conference (EUSIPCO)*, 2005

[GDGN03] Gupta, S. ; Dutt, N. ; Gupta, R. ; Nicolau, A.: SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. In: *Proceedings of the 16th International Conference on VLSI Design (VLSI)*, 2003, pp. 461–466

[GGDN04] Gupta, S. ; Gupta, R. ; Dutt, N. ; Nicolau, A.: *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004

[Golz78] Golze, U.: *Parallele Programmschemata*, Technische Universität Hannover, Habilitation, 1978

[GREA01] Guthaus, M. R. ; Ringenberg, J. S. ; Ernst, D. ; Austin, T. M. ; Mudge, T. ; Brown, R. B.: MiBench: A free, commercially representative embedded benchmark suite. In: *Proceedings of the IEEE International Workshop on Workload Characterization*, 2001, pp. 3–14

[GuHe07] Gu, Y. ; Herbordt, M. C.: High Performance Molecular Dynamics Simulations with FPGA Coprocessors. In: *Proceedings of the Reconfigurable Systems Summer Institute (RSSI)*, 2007

[GuNB08] Guo, Z. ; Najjar, W. ; Buyukkurt, B.: Efficient Hardware Code Generation for FPGAs. In: *ACM Transactions on Architectures and Code Optimization (TACO)* 5 (2008), May, no. 1, pp. 1–26

[GWDL92] Gajski, D. ; Wu, A. ; Dutt, N. ; Lin, S.: *High-Level Synthesis – Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992. – pp. 233–235.

[HaWa97] Hauser, J. R. ; Wawrzynek, J.: Garp: a MIPS processor with a reconfigurable coprocessor. In: *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM)*. Washington, DC, USA : IEEE Computer Society, 1997

[HMSH10] Huthmann, J. ; Müller, P. ; Stock, F. ; Hildenbrand, D. ; Koch, A.: Accelerating High-Level Engineering Computations by Automatic Compilation of Geometric Algebra to Hardware Accelerators. In: *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modelling and Simulation*, 2010

[Hoan93] Hoan, D. T.: Searching Genetic Databases on Splash 2. In: *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE, Apr. 1993, pp. 185–191

[Hone97] Honeywell Technology Center: Versatility Stressmark, Version 0.8 CDRL A001. Minneapolis, USA : Honeywell Technology Center, 1997. – Technical Report

[HTHT09] Hara, Y. ; Tomiyama, H. ; Honda, S. ; Takada, H.: Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. In: *Journal of Information Processing* vol. 17, 2009, pp. 242–254

[Kahn95] Kahn, H. J.: EDIF Version 350/400 and information modelling, 1995, pp. 451

[Kasp05] Kasprzyk, N.: *COMRADE - Ein Hochsprachen-Compiler für adaptive Computersysteme*, Abteilung Entwurf integrierter Schaltungen, Technische Universität Braunschweig, Germany, Diss., Jun. 2005

[KaVK05] Kasprzyk, N. ; Van der Veen, J. ; Koch, A.: Configuration Merging for Adaptive Computer Applications. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2005, pp. 217–222

[Koch04] Koch, A.: *Advances in Adaptive Computer Technology*. Habilitation, Abteilung Entwurf integrierter Schaltungen (E.I.S.), Technische Universität Braunschweig, Germany, Dec. 2004

[KoKa05] Koch, A. ; Kasprzyk, N.: High-Level-Language Compilation for Reconfigurable Computers. In: *Proceedings of the International Conference on Reconfigurable Communication-centric SoCs (ReCoSoC)*, 2005

[LaKo00] Lange, H. ; Koch, A.: Memory Access Schemes for Configurable Processors. In: *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL)*. London, UK : Springer-Verlag, Aug. 2000, pp. 615–625

[LaKo09] Lange, H. ; Koch, A.: Architectures and Execution Models for Hardware/-Software Compilation and their System-Level Realization. In: *IEEE Transactions on Computers* 99, PrePrints (2009)

[LaPM06] Lau, D. ; Pritchard, O. ; Molson, P.: Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions. In: *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Washington, DC, USA : IEEE Computer Society, 2006, pp. 45–56

[Latt02] Lattner, C.: *LLVM: An Infrastructure for Multi-Stage Optimization*, Computer Science Dept., University of Illinois at Urbana-Champaign, Diplomarbeit, Dec. 2002

[LaWK11] Lange, H. ; Wink, T. ; Koch, A.: MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2011

[Leng93] Lengauer, C.: Loop Parallelization in the Polytope Model. In: *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR)*. London, UK : Springer-Verlag, 1993, pp. 398–416

[LePM97] Lee, C. ; Potkonjak, M. ; Mangione-Smith, W. H.: MediaBench: A tool for Evaluating and Synthesizing Multimedia and Communications Systems. In: *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, 1997, pp. 330–335

[LeRK08] Lee, S. ; Raila, D. ; Kindratenko, V.: LLVM-CHiMPS: compilation environment for FPGAs using LLVM compiler infrastructure and CHiMPS computational model. In: *Proceedings of the Reconfigurable Systems Summer Institute (RSSI)*, 2008

[LeRS83] Leiserson, C. E. ; Rose, F. M. ; Saxe, J. B.: Optimizing Synchronous Circuitry by Retiming. In: *Proceedings of the Caltech Conference on VLSI*, 1983, pp. 87–116

[LSKH09] Lange, H. ; Stock, F. ; Koch, A. ; Hildenbrand, D.: Acceleration and Energy Efficiency of a Geometric Algebra Computation Using Reconfigurable Computers and GPUs. In: *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2009, pp. 255–258

[MacM01] MacMillen, D.: *Nimble Compiler Environment for Agile Hardware*. Storming Media LLC (USA), 2001

[NeKo01] Neumann, T. ; Koch, A.: A Generic Library for Adaptive Computing Environments. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)* vol. 2147/2001, 2001, pp. 503–512

[Pana07] Panainte, E. M.: *The Molen Compiler for Reconfigurable Architectures*, Technical University Delft, Diss., Jun. 2007

[PBDM08] Putnam, A. ; Bennett, D. ; Dellinger, E. ; Mason, J. ; Sundararajan, P. ; Eggers, S.: CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 173–178

[PEBD09] Putnam, A. ; Eggers, S. ; Bennett, D. ; Dellinger, E. ; Mason, J. ; Styles, H. ; Sundararajan, P. ; Wittig, R.: Performance and power of cache-based reconfigurable computing. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. New York, NY, USA : ACM, 2009, pp. 281–281

[Pugh91] Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: *Proceedings of the ACM/IEEE Conference on Supercomputing*. New York, NY, USA : ACM, 1991, pp. 4–13

[RaSm94] Razdan, R. ; Smith, M. D.: A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ACM New York, NY, USA, Dec. 1994, pp. 172–180

[Schu07] Schumacher, T. et al.: Accelerating the Cube Cut Problem with an FPGA-augmented Compute Cluster. In: *Parallel Computing: Architectures, Algorithms and Applications*, 2007

[ShHo97] Shapiro, M. ; Horwitz, S.: Fast and accurate flow-insensitive points-to analysis. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, NY, USA : ACM, 1997, pp. 1–14

[ShVu93] Shand, M. ; Vuillemin, J.: Fast implementations of RSA cryptography. In: *Proceedings of the 11th Symposium on Computer Arithmetic*, 1993, pp. 252–259

[SmHo02] Smith, M. D. ; Holloway, G.: An Introduction to machine SUIF and its Portable Libraries for Analysis and Optimization / Division of Engineering and Applied Sciences, Harvard University. 2002. – Technical Report

[SSLM92] Sentovich, E. M. ; Singh, K. J. ; Lavagno, L. ; Moon, C. ; Murgai, R. ; Saldanha, A. ; Savoj, H. ; Stephan, P. R. ; Brayton, R. K. ; Sangiovanni-Vincentelli, A. L.: SIS: A System for Sequential Circuit Synthesis / EECS Department, University of California, Berkeley. 1992 (UCB/ERL M92/41). – Technical Report

[Stee96] Steensgaard, B.: Points-to Analysis in Almost Linear Time. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, NY, USA : ACM, 1996, pp. 32–41

[Stef08] Steffen, C. P.: Nallatech Software Tools on the Maxwell FPGA Parallel Computer. In: *Proceedings of the Reconfigurable Systems Summer Institute (RSSI)*, 2008

[STKY99] Sakurai, R. ; Takahashi, M. ; Kay, A. ; Yamada, A. ; Fujimoto, T. ; Kambe, T.: A scheduling method for synchronous communication in the Bach hardware compiler. In: *Proceedings of the Asian and South Pacific Design Automation Conference (ASP-DAC)* vol. 1, 1999, pp. 193–196

[ThGä10] Thielmann, B. ; Gädke-Lütjens, H.: Modlib: A Flexible Module Library for High-Level Language Compilation / Abteilung Entwurf integrierter Schaltungen (E.I.S.), Technische Universität Braunschweig. 2010. – Technical Report

[UyRB99] Uytterhoeven, G. ; Roose, D. ; Bultheel, A.: Integer Wavelet Transforms using the Lifting Scheme. In: *Proceedings of the International Multiconference on Circuits, Systems, Communications and Computers (CSCC)* vol. 1, 1999, pp. 2651–2653

[Waka99] Wakabayashi, K.: C-based synthesis experiences with a behavior synthesizer, “cyber”. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. New York, NY, USA : ACM, Mar. 1999, pp. 390–393

[Webe08] Weber, N.: *Goto Elimination for the Adaptive Compiler COMRADE*, Embedded Systems and Applications Group (ESA), Technische Universität Darmstadt, in cooperation with Abteilung Entwurf integrierter Schaltungen (E.I.S.), Technische Universität Braunschweig, Germany, Diplomarbeit, Jul. 2008

# Weblinks


[Alti08] Altium Ltd.: *C-to-Hardware Compiler User Manual*. http://www.altium.com/files/altiumdesigner/s08/learningguides/GU0122%20C-to-Hardware%20Compiler%20User%20Manual.pdf, 19th May 2008

[Bass11] Bass, J. L.: *FPGA C Compiler*. http://sourceforge.net/projects/fpgac, 24th Feb. 2011

[Bowe09] Bowen, M. ; Embedded Solutions Ltd.: *Handel-C Language Reference Manual*. http://www.pa.msu.edu/hep/d0/l2/Handel-C/Handel%20C.PDF, 13th Jan. 2009

[Ibmm10] IBM Microelectronics Division: *The PowerPC 440 Core*. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256997006343 1C/\protect\T1\textdollarfile/440_wp.pdf, 17th Nov. 2010

[Impu11] Impulse Accelerated Technologies Inc.: *Impulse-C Website*. http://www.impulseaccelerated.com, Feb. 2011

[Inte11] Intel, Inc.: *Intel Core2 Processor Desktop Processor E6000 Data Sheet*. http://download.intel.com/design/processor/datashts/31327807.pdf, 17th Jan. 2011

[Josh11] Joshi, Jaivardhan: *Parallel Computing and .Net*. http://jai-on-asp.blogspot.com/2010/05/parallel-computing-and-net.html, 24th Feb. 2011

[Ment09] Mentor Graphics Corp.: *Catapult-C data sheet*. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/upload/Catapult_DS_pdf.pdf, 15th Jan. 2009

[Mitr11] Mitrionics: *Mitrion Users' Guide*. http://forum.mitrionics.com/uploads/Mitrion_Users_Guide.pdf, 24th Feb. 2011

[Scal11] Scale Compiler Group: *Scale Compiler Homepage*. http://www.cs.utexas.edu/users/cart/Scale/, 24th Feb. 2011

[Xili10] Xilinx, Inc.: *Virtex-6 Family Overview*. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, 30 Nov. 2010

[Xils10] Xilinx, Inc.: *Virtex-7 Series FPGAs*. http://www.xilinx.com/publications/prod_mktg/Virtex7-Product-Table.pdf, 30th Nov. 2010

# Index

# Acronyms

| | |
|---|---|
| ALU | Arithmetic Logic Unit |
| AST | Abstract Syntax Tree |
| AT | Activate Token |
| BSC | Board Support Components |
| CDFG | Control Data Flow Graph |
| CF | Control Flow Frame |
| CFG | Control Flow Graph |
| COCOMA | COMRADE Controller Micro-Architecture |
| CPP | C Preprocessor |
| CPU | Central Processing Unit |
| CT | Cancel Token |
| DFG | Data Flow Graph |
| ECFG | Line Graph of the CFG |
| EDIF | Electronic Data Interchange Format |
| EE | Early Evaluation |
| FF | Flip-Flop |
| FPGA | Field Programmable Gate Array |
| GLACE | Generic Library for Adaptive Computing Environments |
| HDL | Hardware Description Language |
| HLS | High-Level Synthesis |
| HTG | Hierarchical Task Graph |
| HW | Hardware |
| II | Initiation Interval |
| IR | Intermediate Representation |
| LDP | Loop Data Path |
| LIS | Latency-Insensitive System |
| LPU | Local Paging Unit |
| LUT | Look-Up Table |
| MA | Memory Access |
| MARC | Memory Architecture for Reconfigurable Computers |
| MDF | Meta Data Fetcher |
| NI | Node Input Configuration |

| | |
|---|---|
| NIC | Node Input Control Configuration |
| NID | Node Input Data Configuration |
| NO | Node Output Configuration |
| NOC | Node Output Control Configuration |
| NOD | Node Output Data Configuration |
| pAT | Pseudo Activate Token |
| PDF | Post-Dominance Frontier |
| PPC | PowerPC |
| RCU | Reconfigurable Compute Unit |
| Seq | Sequencer |
| SeqC | Sequencer Configuration |
| SIS | Sequential Interactive Synthesis |
| SSA | Static Single Assignment |
| SUIF | Stanford University Intermediate Format |
| SW | Software |
| UART | Universal Asynchronous Receiver Transmitter |
| ULK | Ultimativer Logik-Konverter |

# A. Sequencer Conditions

This Chapter exactly defines how the COCOMA sequencer computes a node input control configuration $NIC_{t_2}$ and a new sequencer configuration $SeqC_{t_2}$ from a given node output control configuration $NOC_{t_1}$ and an existing sequencer configuration $SeqC_{t_1}$. The node input ports addressed by a *NIC* are listed in the left-hand column of Table A.1. For each input port the right-hand column specifies where in this Chapter the associated **sequencer condition** is located, i.e., a formula defining for each node which value is assigned to the designated input port. Some of the sequencer conditions are dependent on the scheduling type. In such cases, two conditions are given, one for the dynamic CT model and one for static CTs.

To simplify the notation, we omit timing subscripts from now on; the sequencer conditions represent combinatorial logic anyway, thus all signals are assigned the same clock cycle.

Although *NIC* assigns each of the ports an integral value, in practice, most ports are assigned either 0 or 1. Therefore, we use boolean algebra and assign *false* or *true*; formally, however, a *false* assignment stands for 0, and a *true* assignment for 1.

For multi-bit inputs such as the *Start* signal of a mux node, we break down the assignment to single bits by addressing the associated incoming data edges. For example, for a mux node $y$ which has two data predecessors $x_1$ and $x_2$, we assign a boolean value to $Start(x_1, y)$ and $Start(x_2, y)$, which is equivalent to assigning a number between 0 and 3 to $Start(y)$.

Instead of directly defining the new sequencer configuration $SeqC_{t_2}$ for a given $SeqC_{t_1}$, we specify formulae to determine if the *ReadyConsumed* or *CancelConsumed* value of a given edge needs to be set (i.e., a change from 0 to 1) or reset (from 1 to 0). Table A.2 lists the used set and reset signals.

Note that the sequencer conditions use the node neighborhood notation introduced in Section 6.2. Due to the condition complexity, we have moved several subconditions to extra Tables.

| NIC | Definition |
|---|---|
| *Start* | Default: Table A.12; for mux and loopMux inputs: Table A.13 |
| *StartCtrl* | Table A.14 |
| *StartCtrlPseudo* | Table A.15 |
| *Sel* | Table A.17 |
| *StartSel* | Table A.16 |
| *ResultReadyAck* | Table A.26 |
| *Cancel* | Table A.31 |
| *CancelStateAck* | Default, dynCT: Table A.35; for mux and loopMux inputs, dynCT: A.36; default, statCT: Table A.37; for mux and loopMux inputs, statCT: A.38 |
| CancelStateCtrlAck | Table A.40 |

Table A.1.: Left: inputs ports addressed by a *NIC*; right: pointer to the Table defining the sequencer condition.

| SeqC Set/Reset Signal | Definition |
|---|---|
| *SetReadyConsumed* | Table A.44 |
| *ResetReadyConsumed* | Table A.45 |
| *SetCancelConsumed* | Table A.48 |
| *ResetCancelConsumed* | Table A.45 |

Table A.2.: Left: signals signifying a set or reset of a *SeqC* function; right: pointer to the Table defining the sequencer condition.

***AlwActCreateAT*(*y*) :⇔**

| | |
|---|---|
| (1) $y$ has an incoming *alwAct*-annotated control edge, such that | $\exists(x_{con}, y) \in E_{con} : [$<br>$(ann(x_{con}, y) = alwAct) \wedge [$ |
| (2) an AT is created from a violated condition, i.e., | |
| (a) there is a valid result at the controller output, and | $(ResultReady(x_{con}) \wedge$<br>$\neg ReadyConsumed(x_{con}, y) \wedge$ |
| (b) the control condition is not fulfilled, or | $\neg eval(x_{con}, y)) \vee$ |
| (3) an AT is created from a CT, i.e., | |
| (a) $x_{con}$ is canceled, and<br>(b) $y$ has not already consumed that CT. | $(CancelStateControl(x_{con}) \wedge$<br>$\neg CancelConsumed(x_{con}, y))]$ |
| | ] |

Table A.3.: Definition of condition *AlwActCreateAT*; used by condition *Start* (Table A.12).

*SchedType* = *dynCT*: ***DataAT*($x_{dat}$, *y*) :⇔**

| | |
|---|---|
| (1) $x_{dat}$ has a valid result,<br>(2) which has not yet been consumed by $y$, and<br>(3) $y$ is not canceled. | $ResultReady(x_{dat}) \wedge$<br>$\neg ReadyConsumed(x_{dat}, y) \wedge$<br><br>$(t(y) \notin \{mux, loopMux\} \Rightarrow$<br>$\neg CancelState(y)) \wedge$<br>$(t(y) in \{mux, loopMux\} \Rightarrow$<br>$\neg CancelState(x_{dat}, y))$ |

Table A.4.: Definition of condition *DataAT* (dynCT); used by condition *DataDependencesFulfilled* (Table A.6).

*SchedType = statCT*: $\boldsymbol{DataAT(x_{dat}, y) :\Leftrightarrow}$

| | |
|---|---|
| (1) $x_{dat}$ has a valid result,<br>(2) which has not yet been consumed by $y$. | $ResultReady(x_{dat}) \wedge$<br>$\neg ReadyConsumed(x_{dat}, y)$ |

Table A.5.: Definition of condition *DataAT* (statCT); used by condition *DataDependencesFulfilled* (Table A.6).

$\boldsymbol{DataDependencesFulfilled(y) :\Leftrightarrow}$

| | |
|---|---|
| (1) If $y$ is a multiplexer,<br><br>at least one data predecessor edge is active. | $t(y) \in Mux \Rightarrow$<br><br>$\exists x \in X_{dat} : DataAT(x, y)$ |
| $\wedge$ | |
| (2) If $y$ is not a multiplexer,<br><br>all data predecessor edges are active. | $t(y) \notin Mux \Rightarrow$<br><br>$\forall x \in X_{dat} : DataAT(x, y)$ |

Table A.6.: Definition of condition *DataDependencesFulfilled*; used by condition *Start* (Table A.12). This condition is not defined for nodes of type mux or loopMux.

$\boldsymbol{DataDependencesFulfilled(x_{dat}, y) :\Leftrightarrow}$

| | |
|---|---|
| (1) The data predecessor edge is active. | $DataAT(x_{dat}, y)$ |

Table A.7.: Definition of condition *DataDependencesFulfilled* for mux; used by condition *Start* (Table A.12). This condition is only defined for nodes of type mux or loopMux.

***ControlAT*($x_{con}$, $y$) :⇔**

| | |
|---|---|
| (1) *always* nodes steadily emit ATs. | $t(x_{con}) = always$ |
| ∨ | |
| (2) For other nodes: | |
| (a) $x_{con}$ has a valid result, and | $[ResultReady(x_{con}) \wedge$ |
| (b) the control condition is fulfilled, and | $eval(x_{con}, y) \wedge$ |
| (c) the result of $x_{con}$ has not yet been consumed by $y$, or | $\neg ReadyConsumed(x_{con}, y)] \vee$ |
| (d) $(x_{con}, y)$ has an *atOnCancel* annotation, and | $[ann(x_{con}, y) = atOnCancel \wedge$ |
| (e) $x_{con}$ is canceled, and | $CancelStateCtrl(x_{con}) \wedge$ |
| (f) that CT has not been consumed yet by $y$. | $\neg CancelConsumed(y)]$ |

Table A.8.: Definition of condition *ControlAT*; used by condition *ControlDependencesFulfilled* (Table A.9).

***ControlDependencesFulfilled*($y$) :⇔**

| | |
|---|---|
| (1) If $y$ is an *all* node, | $t(y) = all \Rightarrow$ |
| all non-nAT control predecessor are active. | $\forall x_{con} \in X_{con} \setminus X_{con,nAT}$ : $ControlAT(x_{con}, y)$ |
| ∧ | |
| (2) Otherwise, | $t(y) \neq all \Rightarrow$ |
| at least one (if existing) non-nAT control predecessor edge is active. | $[X_{con} \setminus X_{con,nAT} \neq \emptyset \Rightarrow$ $\exists x_{con} \in X_{con} \setminus X_{con,nAT}$ : $ControlAT(x_{con}, y)]$ |

Table A.9.: Definition of condition *ControlDependencesFulfilled*; used by condition *Start* (Table A.12).

***MemoryAT*($x_{mem}, y$) :⇔**

| | |
|---|---|
| (1) $x_{mem}$ has a valid result, and<br>(2) the result of $x_{mem}$ has not yet been consumed by $y$. | $ResultReady(x_{mem}) \wedge$<br>$\neg ReadyConsumed(x_{mem}, y)$ |

Table A.10.: Definition of condition *MemoryAT*; used by condition *MemoryDependencesFulfilled* (Table A.11).

***MemoryDependencesFulfilled*($y$) :⇔**

| | |
|---|---|
| (1) If $y$ is an *all* node,<br><br>all memory predecessor edges are active. | $t(y) = all \Rightarrow$<br><br>$\forall x_{mem} \in X_{mem}$ :<br>$MemoryAT(x_{mem}, y)$ |
| $\wedge$ | |
| (2) Otherwise,<br><br>at least one (if existing) memory predecessor edge is active. | $t(y) \neq all \Rightarrow$<br><br>$[X_{mem} \neq \emptyset \Rightarrow$<br>$\exists x_{mem} \in X_{mem}$ :<br>$MemoryAT(x_{mem}, y)]$ |

Table A.11.: Definition of condition *MemoryDependencesFulfilled*; used by condition *Start* (Table A.12).

***Start*(*y*) :⇔**

| | |
|---|---|
| (1) Initial nodes are never activated through COCOMA-logic; | $t(y) \neq initial$ |
| | $\wedge$ |
| (2) data dependences are fulfilled, and | $DataDependencesFulfilled(y)$ |
| | $\wedge$ |
| (3) memory dependences are fulfilled, or (in the statCT case) $y$ is canceled. | $MemoryDependencesFulfilled(y) \vee$<br><br>$(SchedType = statCT \wedge$<br>$CancelState(y))$ |

Table A.12.: Definition of condition *Start*. (1) Initial nodes are never activated by COCOMA data/control/memory predecessors, but they *are* (re-)activated when the associated software-to-hardware transition takes place (this is not integrated in the sequencer formalism). This condition is not defined for multiplexer nodes.

***Start*($x_{dat}$, $y$) :⇔**

| | |
|---|---|
| (1) Data dependences are fulfilled. | $DataDependencesFulfilled(x_{dat}, y)$ |

Table A.13.: Definition of condition *Start* for incoming data edges of multiplexer nodes.

***StartCtrl*(*y*) :⇔**

| | |
|---|---|
| (1) Control dependences are fulfilled, or | $ControlDependencesFulfilled(y) \vee$ |
| (2) $y$ is activated by an incoming *alwAct*-annotated edge. | $AlwActCreateAT(y)$ |

Table A.14.: Definition of condition *StartCtrl*. This condition is not defined for nodes of type mux.

***StartCtrlPseudo*(*y*) :⇔**

| | |
|---|---|
| (1) $y$ is activated by an incoming *alwAct* edge. | $AlwActCreateAT(y)$ |
| | $\vee$ |
| (2)<br>(a) $y$ has an incoming CT-capable control edge,<br>(b) whose associated controller is canceled. | $\exists x_{con} \in X_{con,1} \cup X_{con,nAT} \cup X_{con,atOnCancel} :$<br>$CancelStateCtrl(x_{con})$ |

Table A.15.: Definition of condition *StartCtrlPseudo*. This condition is not defined for nodes of type mux.

***StartSel*(*y*) :⇔**

| | |
|---|---|
| There is a control predecessor $x_{con}$ of $y$, such that | $\exists x_{con} \in X_{con}$ : |
| | |
| (1) $x_{con}$ has a valid result, or<br>(2) $x_{con}$ is canceled, but | $(ResultReady(x_{con}) \vee$<br>$CancelStateCtrl(x_{con})) \wedge$ |
| (3) the token (AT or CT) from $x_{con}$ has not yet been consumed. | $(\neg ReadyConsumed(x_{con}, y) \wedge$<br>$\neg CancelConsumed(x_{con}, y))$ |

Table A.16.: Definition of condition *StartSel*. This condition is only defined for nodes of type mux.

$\boldsymbol{Sel(x_{dat}, y) :\Leftrightarrow}$

| | |
|---|---|
| (1) $x_{dat}$ has a valid result, and | $ResultReady(x_{dat})$ |
| | $\wedge$ |
| (2) there is a control predecessor $x_{con}$ of $y$, such that the evaluation bit associated to $x_{dat}$ is true. | $\exists x_{con} \in X_{con}$ : $eval(x_{con}, y)\|_i, i = muxPred_y(x_{dat})$ |

Table A.17.: Definition of condition *Sel*. This condition is only defined for nodes of type mux. (2) Recall that $eval(x_{con}, y)$ is multi-dimensional if $y$ is of type mux. The operator $|_i$ returns the $i$th dimension.

*SchedType = dynCT*: $\boldsymbol{DataSuccessorAccepted(y, z) :\Leftrightarrow}$

| | |
|---|---|
| (1) The *ResultReady* of $y$ has already been consumed by $z$, or | $ReadyConsumed(y, z) \vee$ |
| (2) $z$ has been started already, or | $((t(z) \notin \{mux, loopMux\} \Rightarrow StartAck(z)) \wedge$ $(t(z) \in \{mux, loopMux\} \Rightarrow StartAck(y, z))) \vee$ |
| (3) the edge has not yet propagated a CT from $z$, and | $(\neg CancelConsumed(y, z) \wedge$ |
| (4) $z$ is canceled and | $(t(z) \notin \{mux, loopMux\} \Rightarrow (CancelState(z) \wedge CancelStateAck(z))) \wedge$ |
| (5) its cancel state is being acknowledged. | $(t(z) \in \{mux, loopMux\} \Rightarrow (CancelState(y, z) \wedge CancelStateAck(y, z))))$ |

Table A.18.: Definition of condition *DataSuccessorAccepted* (dynCT); used by condition *DataSuccessorsAccepted* (Table A.20).

*SchedType = statCT*: ***DataSuccessorAccepted*(*y*, *z*) :⇔**

| | |
|---|---|
| (1) The *ResultReady* of $y$ has already been consumed by $z$, or<br>(2) $z$ has been started already. | $ReadyConsumed(y, z) \vee$<br><br>$((t(z) \notin \{mux, loopMux\} \Rightarrow$<br>$StartAck(z)) \wedge$<br>$(t(z) \in \{mux, loopMux\} \Rightarrow$<br>$StartAck(y, z)))$ |

Table A.19.: Definition of condition *DataSuccessorAccepted* (statCT); used by condition *DataSuccessorsAccepted* (Table A.20).

***DataSuccessorsAccepted*(*y*) :⇔**

| | |
|---|---|
| All data successors accept the *ResultReady* of $y$. | $\forall z_{dat} \in Z_{dat}$ :<br>$DataSuccessorAccepted(y, z_{dat})$ |

Table A.20.: Definition of condition *DataSuccessorsAccepted*; used by condition *ResultReadyAck* (Table A.26).

***MemorySuccessorAccepted*(*y*, $z_{mem}$) :⇔**

| | |
|---|---|
| (1) The *ResultReady* of $y$ has been consumed by $z_{mem}$, or<br>(2) $z_{mem}$ is started currently. | $ReadyConsumed(y, z_{mem}) \vee$<br><br>$StartAck(z_{mem})$ |

Table A.21.: Definition of condition *MemorySuccessorAccepted*; used by condition *MemorySuccessorsAccepted* (Table A.22).

***MemorySuccessorsAccepted*(*y*) :⇔**

| | |
|---|---|
| All memory successors accept the *ResultReady* of $y$. | $\forall z_{mem} \in Z_{mem}$ :<br>$MemorySuccessorAccepted(y, z_{mem})$ |

Table A.22.: Definition of condition *MemorySuccessorsAccepted*; used by condition *ResultReadyAck* (Table A.26).

*SchedType = dynCT*: ***ControlSuccessorAccepted*($y, z_{con}$) :⇔**

| | |
|---|---|
| (1) If the control edge has a 1 or *atOnCancel* annotation, | $ann(y, z_{con}) \in \{1, atOnCancel\} \Rightarrow [$ |
| (a) the *ResultReady* of $y$ has already been consumed by $z_{con}$, or | $ReadyConsumed(y, z_{con}) \vee$ |
| (b) $z_{con}$ accepts an AT, or | $StartCtrlAck(z_{con}) \vee$ |
| (c) $z_{con}$ is canceled currently and | $(Cancel(z_{con}) \wedge$ |
| (d) accepts the cancel. | $CancelAck(z_{con}))]$ |
| | $\wedge$ |
| (2) If the control edge has an *nCT* annotation, | $ann(y, z_{con}) = nCT \Rightarrow [$ |
| (a) the *ResultReady* of $y$ has already been consumed by $z_{con}$, or | $ReadyConsumed(y, z_{con}) \vee$ |
| (b) $z_{con}$ is started now, or | $StartControlAck(z_{con}) \vee$ |
| (c) $y$ has a valid result and | $(ResultReady(y) \wedge$ |
| (d) the control condition for $(y, z_{con})$ is violated. | $\neg eval(y, z_{con}))]$ |
| | $\wedge$ |
| (3) If the control edge has an *nAT* annotation, | $ann(y, z_{con}) = nAT \Rightarrow [$ |
| (a) $z_{con}$ is canceled now and | $(Cancel(z_{con}) \wedge$ |
| (b) accepts the cancel, or | $CancelAck(z_{con})) \vee$ |
| (c) $y$ has a valid result and | $(ResultReady(y) \wedge$ |
| (d) the control condition is fulfilled. | $eval(y, z_{con}))]$ |
| | $\wedge$ |
| (4) If the control edge has an *alwAct* annotation, | $ann(y, z_{con}) = alwAct \Rightarrow [$ |
| (a) the *ResultReady* of $y$ has already been consumed by $z_{con}$, or | $ReadyConsumed(y, z_{con}) \vee$ |
| (b) $z_{con}$ is started now, or | $StartControlAck(z_{con}) \vee$ |
| (c) $z_{con}$ is canceled, and | $(CancelState(z_{con}) \wedge$ |
| (d) that CT is now extinguished. | $CancelStateAck(z_{con}))]$ |

*The Table is continued on the next page.*

| | |
|---|---|
| | $\wedge$ |
| (5) If the control edge has a *muxControl* annotation,<br>(a) $z_{con}$ accepts the token, or<br>(b) has already consumed the token. | $ann(y, z_{con}) = muxControl \Rightarrow [$<br><br>$StartSelAck(z_{con}) \vee$<br>$ReadyConsumed(y, z_{con})]$ |

Table A.23.: Definition of condition *ControlSuccessorAccepted* (dynCT); used by condition *ControlSuccessorsAccepted* (Table A.25).

*SchedType = statCT*: $\boldsymbol{ControlSuccessorAccepted(y, z_{con}) :\Leftrightarrow}$

| | |
|---|---|
| (1) If the control edge has a 1 or *atOnCancel* annotation,<br>(a) the *ResultReady* of $y$ has already been consumed by $z_{con}$, or<br>(b) $z_{con}$ is started now, or<br>(c) $z_{con}$ is canceled currently and<br>(d) accepts the cancel. | $ann(y, z_{con}) \in \{1, atOnCancel\} \Rightarrow [$<br><br>$ReadyConsumed(y, z_{con}) \vee$<br><br><br>$StartControlAck(z_{con}) \vee$<br>$(Cancel(z_{con}) \wedge$<br><br>$CancelAck(z_{con}))]$ |
| | $\wedge$ |
| (2) If the control edge has an *nCT* annotation,<br>(a) the *ResultReady* of $y$ has already been consumed by $z_{con}$, or<br>(b) $z_{con}$ is started now, or<br>(c) $y$ has a valid result and<br>(d) the control condition for $(y, z_{con})$ is violated. | $ann(y, z_{con}) = nCT \Rightarrow [$<br><br>$ReadyConsumed(y, z_{con}) \vee$<br><br><br>$StartControlAck(z_{con}) \vee$<br>$(ResultReady(y) \wedge$<br>$\neg eval(y, z_{con}))]$ |
| | $\wedge$ |

*The Table is continued on the next page.*

| | |
|---|---|
| (3) If the control edge has an *nAT* annotation,<br>(a) $z_{con}$ is canceled now and<br>(b) accepts the cancel, or<br>(c) $y$ has a valid result and<br>(d) the control condition for $(y, z_{con})$ is fulfilled. | $ann(y, z_{con}) = nAT \Rightarrow [$<br><br>$(Cancel(z_{con}) \wedge$<br>$CancelAck(z_{con})) \vee$<br>$(ResultReady(y) \wedge$<br>$eval(y, z_{con}))]$ |
| | $\wedge$ |
| (4) If the control edge has an *alwAct* annotation,<br>(a) the *ResultReady* of $y$ has already been consumed by $z_{con}$, or<br>(b) $z_{con}$ is started now. | $ann(y, z_{con}) = alwAct \Rightarrow [$<br><br>$ReadyConsumed(y, z_{con}) \vee$<br><br>$StartControlAck(z_{con})]$ |
| | $\wedge$ |
| (5) If the control edge has a *muxControl* annotation,<br>(a) $z_{con}$ accepts the token, or<br>(b) has already consumed the token. | $ann(y, z_{con}) = muxControl \Rightarrow [$<br><br>$StartSelAck(z_{con}) \vee$<br>$ReadyConsumed(y, z_{con})]$ |

Table A.24.: Definition of condition *ControlSuccessorAccepted* (statCT); used by condition *ControlSuccessorsAccepted* (Table A.25).

***ControlSuccessorsAccepted*(y) :⇔**

| | |
|---|---|
| All control successors acknowledge the ResultReady or Cancel. | $\forall z_{con} \in Z_{con}$ :<br>$ControlSuccessorAccepted(y, z_{con})$ |

Table A.25.: Definition of condition *ControlSuccessorsAccepted*; used by condition *ResultReadyAck* (Table A.26).

***ResultReadyAck*(*y*) :⇔**

| | |
|---|---|
| (1) If $y$ does *not* have outgoing control edges,<br>(a) data successors and<br>(b) memory successors<br>acknowledge the *ResultReady*. | $Z_{con} = \emptyset \Rightarrow$<br><br>$DataSuccessorsAccepted(y) \wedge$<br>$MemorySuccessorsAccepted(y)$ |
| $\wedge$ | |
| (2) If $y$ *has* an outgoing control edge,<br>its control successors acknowledge the *ResultReady*. | $Z_{con} \neq \emptyset \Rightarrow$<br><br>$ControlSuccessorsAccepted(y)$ |

Table A.26.: Definition of condition *ResultReadyAck*.

*SchedType* = *dynCT*: ***DataCT*(*y*, $z_{dat}$) :⇔**

| | |
|---|---|
| (1) $y$ does not have a valid result, and | $\neg ResultReady(y) \wedge$ |
| (2) $z_{dat}$ has a CT, | $(t(z_{dat}) \notin \{mux, loopMux\} \Rightarrow$<br>$CancelState(z_{dat})) \wedge$<br>$(t(z_{dat}) \in \{mux, loopMux\} \Rightarrow$<br>$CancelState(y, z_{dat})) \wedge$ |
| (3) which has not yet been consumed by $y$. | $\neg CancelConsumed(y, z_{dat})$ |

Table A.27.: Definition of condition *DataCT* (dynCT), used by condition *Cancel* (Table A.31).

***CancelThroughIncomingControlEdge*(*y*) :⇔**

| | |
|---|---|
| (1) $y$ has an incoming *atOnCancel* control edge, and | $x \in X_{con,atOnCancel} \wedge$ |
| (a) $x_{con}$ has a valid result, | $ResultReady(x_{con}) \wedge$ |
| (b) which has not yet been consumed, and | $\neg ReadyConsumed(x_{con}, y) \wedge$ |
| (c) its control condition to $y$ is violated. | $\neg eval(x_{con}, y)$ |
| ∨ | |
| (2) $y$ has another incoming CT-capable control edge, and | $X_{con} \setminus (X_{con,nCT} \cup X_{con,alwAct} \cup X_{con,atOnCancel}) \neq \emptyset \wedge$ |
| (a) $x_{con}$ has a valid result, | $[ResultReady(x_{con}) \wedge$ |
| (b) which has not yet been consumed, and | $\neg ReadyConsumed(x_{con}, y) \wedge$ |
| (c) its control condition to $y$ is violated, or | $\neg eval(x_{con}, y)] \vee$ |
| (d) $x_{con}$ is canceled, and | $[CancelStateCtrl(x_{con}) \wedge$ |
| (e) the cancel has not yet been consumed by $y$. | $\neg CancelConsumed(x_{con}, y)]$ |

Table A.28.: Definition of condition *CancelThroughIncomingControlEdge*; used by condition *Cancel* (Table A.31). This condition is not defined for nodes of type mux.

*SchedType = dynCT*: ***CancelThroughOutgoingDataEdges*(*y*) :⇔**

| | |
|---|---|
| (1) $y$ has a data successor, and | $Z_{dat} \neq \emptyset \wedge$ |
| (2) $y$ has no incoming, CT-capable control edge, or | $[X_{con,1} \cup X_{con,nAT} \cup X_{con,atOnCancel} = \emptyset \vee$ |
| (3) $y$ has exactly 1 incoming control edge which comes from an *initial* or *always* node, and | $(\lvert X_{con}\rvert = 1 \wedge x_{con} \in X_{con} :$ $t(x_{con}) \in always, initial)] \wedge$ |
| (4) each data successor has a CT, and | $\forall z_{dat} \in Z_{dat} : DataCT(z_{dat}) \wedge$ |
| (5) $y$ is not a multiplexer. | $t(y) \notin Mux$ |

Table A.29.: Definition of condition *CancelThroughOutgoingDataEdges* (dynCT); used by condition *Cancel* (Table A.31).

*SchedType = statCT*: ***CancelThroughOutgoingDataEdges*(*y*) :⇔**

| | |
|---|---|
| This condition is always false, because in static CT mode, CTs are not propagated along data edges. | *false* |

Table A.30.: Definition of condition *CancelThroughOutgoingDataEdges* (statCT); used by condition *Cancel* (Table A.31).

***Cancel*(*y*) :⇔**

| | |
|---|---|
| (1) If $y$ is not a mux and not an irqDataMux, | $t(y) \notin \{mux, irqDataMux\} \Rightarrow$ |
| (a) $y$ is canceled through an incoming control edge, | [*CancelThroughIncoming-ControlEdge*($y$)∧ |
| (b) where all ATs of $y$ are consumed (except for loop-Muxes), or | (¬*ResultReady*($y$)∨ $t(y) = loopMux$)]∨ |
| (c) through outgoing data edges. | *CancelThroughOutgoing-DataEdges*($y$) |
| ∧ | |
| (2) Otherwise, $y$ is never canceled. | $t(y) \in \{mux, irqDataMux\} \Rightarrow$ *false* |

Table A.31.: Definition of condition *Cancel*.

*SchedType* = *dynCT*:
***DataPredCancelStateResetLocalStandard*($x_{dat}$, $y$) :⇔**

| | |
|---|---|
| (1) $x_{dat}$ is canceled now and | (*Cancel*($x_{dat}$)∧ |
| (2) accepts the cancel, or | *CancelAck*($x_{dat}$))∨ |
| (3) $y$ has already consumed $x_{dat}$'s cancel, or | *CancelConsumed*($x_{dat}$, $y$)∨ |
| (4) $x_{dat}$ has a valid result, and | (*ResultReady*($x_{dat}$)∧ |
| (5) $y$ has a cancel, and | $(t(y) \notin \{mux, loopMux\} \Rightarrow$ *CancelState*($y$))∧ |
| | $(t(y) \in \{mux, loopMux\} \Rightarrow$ *CancelState*($x_{dat}$, $y$))∧ |
| (6) neither CT | ¬*CancelConsumed*($x_{dat}$, $y$)∧ |
| (7) nor AT have already been consumed by the other party. | ¬*ReadyConsumed*($x_{dat}$, $y$)) |

Table A.32.: Definition of condition *DataPredCancelStateResetLocalStandard* (dynCT), used by condition *CancelStateAck* (Table A.35).

*SchedType = dynCT*:
***DataPredCancelStateResetLocalMux*($x_{dat}$, $y$) :⇔**

| | |
|---|---|
| (1) $x_{dat}$ has a valid result, and<br>(2) $y$ has a cancel, and<br><br><br><br>(3) neither CT<br>(4) nor AT have already been consumed by the other party. | $(ResultReady(x_{dat}) \wedge$<br>$(t(y) \notin \{mux, loopMux\} \Rightarrow$<br>$CancelS\,tate(y)) \wedge$<br>$(t(y) \in \{mux, loopMux\} \Rightarrow$<br>$CancelState(x_{dat}, y)) \wedge$<br>$\neg CancelConsumed(x_{dat}, y) \wedge$<br>$\neg ReadyConsumed(x_{dat}, y))$ |

Table A.33.: Definition of condition *DataPredCancelStateResetLocalMux* (dynCT), used by condition *CancelStateAck* (Table A.36).

*SchedType = dynCT*: ***DataPredCancelStateResetLocal*($x_{dat}$, $y$) :⇔**

| | |
|---|---|
| (1) If $x_{dat}$ is a multiplexer,<br>(2) *DataPredCancelState-ResetLocalMux* is fulfilled. | $t(x_{dat}) \in Mux \Rightarrow$<br>*DataPredCancelStateReset-LocalMux*($x_{dat}$, $y$) |
| $\wedge$ | |
| (1) Otherwise,<br>(2) *DataPredCancelState-ResetLocalStandard* is fulfilled. | $t(x_{dat}) \notin Mux \Rightarrow$<br>*DataPredCancelStateResetLocal-Standard*($x_{dat}$, $y$) |

Table A.34.: Definition of condition *DataPredCancelStateResetLocal* (dynCT), used by condition *CancelStateAck* (Table A.35).

*SchedType = dynCT*: ***CancelStateAck*($y$) :⇔**

| | |
|---|---|
| $y$'s *CancelState* is reset, if<br>(1) all data predecessors accept the cancel. | $(\forall x_{dat} \in X_{dat}$ :<br>*DataPredCancelStateReset-Local*($x_{dat}$, $y$)) |

Table A.35.: Definition of condition *CancelStateAck* (dynCT). Not defined for nodes of type mux or loopMux.

*SchedType = dynCT*: ***CancelStateAck*($x_{dat}$, $y$) :⇔**

| | |
|---|---|
| $y$'s CancelState is reset, if the connected predecessor accepts the CT. | *DataPredCancelStateReset-Local*($x_{dat}$, $y$)) |

Table A.36.: Definition of condition *CancelStateAck* (dynCT) for nodes of type mux or loopMux.

*SchedType = statCT*: ***CancelStateAck*($y$) :⇔**

| | |
|---|---|
| $y$'s CancelState is reset, if (1) data inputs are consumed now. | *StartAck*($y$) |

Table A.37.: Definition of condition *CancelStateAck* (statCT). Not defined for nodes of type mux or loopMux.

*SchedType = statCT*: ***CancelStateAck*($x_{dat}$, $y$) :⇔**

| | |
|---|---|
| $y$'s CancelState is never true, thus the acknowledgment is always false. | *false* |

Table A.38.: Definition of condition *CancelStateAck* (statCT) for nodes of type mux or loopMux.

***ControlSuccCancelStateResetLocal*$(y, z_{con})$ :⇔**

| | |
|---|---|
| (1) If $(y, z_{con})$ is annotated with 1 or *nAT*, and $z_{con}$ is not a *mux*, | $ann(y, z_{con}) \in \{1, nAT\} \wedge$<br>$t(z_{con}) \neq mux \Rightarrow$ |
| (a) $z_{con}$ is canceled now and<br>(b) accepts the cancel, or | $(Cancel(z_{con}) \wedge$<br>$CancelAck(z_{con})) \vee$ |
| (c) $z_{con}$ has already consumed $y$'s cancel. | $CancelConsumed(y, z_{con})$ |
| ∧ | |
| (2) If $(y, z_{con})$ is annotated with *atOnCancel*, and $z_{con}$ is not a *mux*, | $ann(y, z_{con}) = atOnCancel \wedge$<br>$t(z_{con}) \neq mux \Rightarrow$ |
| (a) $z_{con}$ is canceled now and<br>(b) accepts the cancel, or | $(Cancel(z_{con}) \wedge$<br>$CancelAck(z_{con})) \vee$ |
| (c) $y$ is canceled, and<br>(d) $z_{con}$ receives an AT, and<br>(e) accepts it, or | $(CancelStateCtrl(y) \wedge$<br>$StartCtrl(z_{con}) \wedge$<br>$StartCtrlAck(z_{con})) \vee$ |
| (f) $z_{con}$ has already consumed $y$'s cancel. | $CancelConsumed(y, z_{con})$ |
| ∧ | |
| (3) If $(y, z_{con})$ is annotated with *nCT*,<br>(a) $y$'s cancel just disappears. | $ann(y, z_{con}) = nCT \Rightarrow$<br>$true$ |
| ∧ | |

*The Table is continued on the next page.*

| | |
|---|---|
| (4) If $(y, z_{con})$ is annotated with *alwAct*, | $ann(y, z_{con}) = alwAct \Rightarrow$ |
| (a) $z_{con}$ receives a control AT, or<br>(b) has already received it, or | $[StartCtrlAck(z_{con}) \vee$<br>$ReadyConsumed(y, z_{con})] \vee$ |
| (c) $z_{con}$ has a CT<br>(d) which is erased now. | $[CancelState(z_{con}) \wedge$<br>$CancelStateAck(z_{con})]$ |
| | $\wedge$ |
| (5) If $z_{con}$ is a mux, | $t(z_{con}) = mux \Rightarrow$ |
| (a) $z_{con}$ accepts the select signal, or | $StartSelAck(z_{con}) \vee$ |
| (b) $z_{con}$ has already consumed the select. | $CancelConsumed(y, z_{con})$ |

Table A.39.: Definition of condition *ControlSuccCancelStateResetLocal*; used by condition *CancelStateCtrlAck* (Table A.40).

***CancelStateCtrlAck*(*y*) :⇔**

| | |
|---|---|
| $y$'s CancelStateCtrl is acknowledged, if<br>(1) all control successors accept the cancel. | $(\forall z_{con} \in Z_{con}$ :<br>*ControlSuccCancelStateReset-Local*$(y, z_{con}))$ |

Table A.40.: Definition of condition *CancelStateCtrlAck*.

***SetReadyConsumedStandard*$(y, z)$ :⇔**

| | |
|---|---|
| (1) If $(y, z)$ is a data edge, | $(y, z) \in E_{dat} \Rightarrow$ |
| (a) $y$ has an AT, | $ResultReady(y) \wedge$ |
| (b) which has not already been acknowledged, and | $\neg ResultReadyAck(y) \wedge$ |
| (c) either $z$ consumes the AT, or | $(StartAck(z) \vee$ |
| (d) the AT is extinguished along the edge by a CT from $z$ (only in dynCT mode). | $(SchedType = dynCT \wedge$<br>$CancelState(z) \wedge$<br>$CancelStateAck(z)))$ |
| | $\wedge$ |
| (2) If $(y, z)$ is a control edge, | $(y, z) \in E_{con} \Rightarrow$ |
| (a) $y$ has an AT, | $ResultReady(y) \wedge$ |
| (b) which has not already been acknowledged, and | $\neg ResultReadyAck(y) \wedge$ |
| (c) either $z$ consumes the AT via *StartCtrl*, or | $[StartCtrlAck(z) \vee$ |
| (d) the AT is turned into a CT, consumed by $z$. | $(Cancel(z) \wedge$<br>$CancelAck(z))]$ |
| | $\wedge$ |
| (3) If $(y, z)$ is a memory edge, | $(y, z) \in E_{mem} \Rightarrow$ |
| (a) $y$ has an AT, | $ResultReady(y) \wedge$ |
| (b) which has not already been acknowledged, and | $\neg ResultReadyAck(y) \wedge$ |
| (c) $z$ consumes the AT. | $StartAck(z)$ |

Table A.41.: Definition of condition *SetReadyConsumedStandard*. Not defined for edges leading to nodes of type mux or loopMux.

***SetReadyConsumedMux*(*y*, *z*) :⇔**

| | |
|---|---|
| (1) If $(y, z)$ is a data edge, | $(y, z) \in E_{dat} \Rightarrow$ |
| (a) $y$ has an AT,<br>(b) which has not already been acknowledged, and<br>(c) either $z$ consumes the AT, or<br>(d) the AT is extinguished on the edge by a CT from $z$ (only in dynCT mode). | $ResultReady(y) \wedge$<br>$\neg ResultReadyAck(y) \wedge$<br>$(StartAck(y, z) \vee$<br>$(SchedType = dynCT \wedge$<br>$CancelState(y, z) \wedge$<br>$CancelStateAck(y, z)))$ |
| $\wedge$ | |
| (2) If $(y, z)$ is a control edge, | $(y, z) \in E_{con} \Rightarrow$ |
| (a) $y$ has an AT,<br>(b) which has not already been acknowledged, and<br>(c) $z$ consumes the select signal. | $ResultReady(y) \wedge$<br>$\neg ResultReadyAck(y) \wedge$<br>$StartSelAck(z)$ |

Table A.42.: Definition of condition *SetReadyConsumedMux*. Only defined for edges leading to nodes of type mux.

***SetReadyConsumedLoopMux*$(y, z)$ :⇔**

| | |
|---|---|
| (1) If $(y, z)$ is a data edge, | $(y, z) \in E_{dat} \Rightarrow$ |
| (a) $y$ has an AT,<br>(b) which has not already been acknowledged, and | *ResultReady*$(y)\wedge$<br>$\neg$*ResultReadyAck*$(y)\wedge$ |
| (c) either $z$ consumes the AT, or | (*StartAck*$(y, z)\vee$ |
| (d) the AT is extinguished on the edge by a CT from $z$ (only in dynCT mode). | (*SchedType* = *dynCT*$\wedge$<br>*CancelState*$(z)\wedge$<br>*CancelStateAck*$(z)$)) |
| $\wedge$ | |
| (2) If $(y, z)$ is a control edge, | $(y, z) \in E_{con} \Rightarrow$ |
| (a) $y$ has an AT,<br>(b) which has not already been acknowledged, and | *ResultReady*$(y)\wedge$<br>$\neg$*ResultReadyAck*$(y)\wedge$ |
| (c) either $z$ consumes the AT via *StartCtrl*, or | [*StartCtrlAck*$(z)\vee$ |
| (d) the AT is turned into a CT, consumed by $z$. | (*Cancel*$(z)\wedge$<br>*CancelAck*$(z)$)] |

Table A.43.: Definition of condition *SetReadyConsumedLoopMux*. Only defined for edges leading to nodes of type loopMux.

***SetReadyConsumed*$(y, z)$ :⇔**

| | |
|---|---|
| (1) If $z$ is neither a mux nor a loopMux, *SetReadyConsumedStandard* is fulfilled. | $t(z) \notin \{mux, loopMux\} \Rightarrow$<br>*SetReadyConsumedStandard*$(y, z)$ |
| $\wedge$ | |
| (2) If $z$ is a mux, *SetReadyConsumedMux* is fulfilled. | $t(z) = mux \Rightarrow$<br>*SetReadyConsumedMux*$(y, z)$ |
| $\wedge$ | |
| (3) If $z$ is a loopMux, *SetReadyConsumedLoopMux* is fulfilled. | $t(z) = loopMux \Rightarrow$<br>*SetReadyConsumedLoopMux*$(y, z)$ |

Table A.44.: Definition of condition *SetReadyConsumed*.

***ResetReadyConsumed*(*y*, *z*) :⇔**

| | |
|---|---|
| (1) *y* is notified a *ResultReadyAck*. | *ResultReadyAck*(*y*) |

Table A.45.: Definition of condition *ResetReadyConsumed*.

***SetCancelConsumedStandard*$(y, z)$ :⇔**

| | |
|---|---|
| (1) If $(y, z)$ is a data edge, | $(y, z) \in E_{dat} \Rightarrow$ |
| (a) $z$ emits a CT,<br>(b) which has not already been acknowledged, and<br>(c) either $y$ consumes the CT (and $y$ is not a multiplexer), or<br>(d) the CT is extinguished along the edge by an AT from $y$. | $CancelState(z) \wedge$<br>$\neg CancelStateAck(z) \wedge$<br>$[(Cancel(y) \wedge$<br>$CancelAck(y) \wedge$<br>$t(y) \notin Mux) \vee$<br>$(ResultReady(y) \wedge$<br>$\neg ResultReadyAck(y))]$ |
| $\wedge$ | |
| (2) If $(y, z)$ is a control edge with *alwAct* or *atOnCancel* annotation, | $(y, z) \in E_{con,alwAct} \Rightarrow$ |
| (a) $y$ has a CT,<br>(b) which has not yet been acknowledged, and<br>(c) $z$ consumes the AT which has been generated at the edge from $y$'s CT. | $CancelStateCtrl(y) \wedge$<br>$\neg CancelStateCtrlAck(y) \wedge$<br>$StartCtrl(z) \wedge$<br>$StartCtrlAck(z)$ |
| $\wedge$ | |
| (3) If $(y, z)$ is a control edge having neither an *atOnCancel*, nor an *alwAct* annotation, | $(y, z) \in E_{con} \setminus (E_{con,alwAct} \cup E_{con,atOnCancel}) \Rightarrow$ |
| (a) $y$ has a CT,<br>(b) which has not yet been acknowledged, and<br>(c) $z$ consumes the CT. | $CancelStateCtrl(y) \wedge$<br>$\neg CancelStateCtrlAck(y) \wedge$<br>$Cancel(z) \wedge$<br>$CancelAck(z)$ |

Table A.46.: Definition of condition *SetCancelConsumedStandard*. Not defined for edges leading to nodes of type mux or loopMux.

***SetCancelConsumedLoopMux*(*y*, *z*) :⇔**

| | |
|---|---|
| (1) If $(y, z)$ is a data edge, | $(y, z) \in E_{dat} \Rightarrow$ |
| (a) $z$ emits a CT,<br>(b) which has not already been acknowledged, and<br>(c) either $y$ consumes the CT (and $y$ is not a multiplexer), or<br>(d) the CT is extinguished at the edge by an AT from $y$. | $CancelState(y, z) \wedge$<br>$\neg CancelStateAck(y, z) \wedge$<br>$[(Cancel(y) \wedge$<br>$CancelAck(y) \wedge$<br>$t(y) \notin Mux) \vee$<br>$(ResultReady(y) \wedge$<br>$\neg ResultReadyAck(y))]$ |
| $\wedge$ | |
| (2) If $(y, z)$ is a control edge, | $(y, z) \in E_{con} \Rightarrow$ |
| (a) $y$ has a CT,<br>(b) which has not yet been acknowledged, and<br>(c) $z$ consumes the CT. | $CancelStateCtrl(y) \wedge$<br>$\neg CancelStateCtrlAck(y) \wedge$<br>$Cancel(z) \wedge$<br>$CancelAck(z)$ |

Table A.47.: Definition of condition *SetCancelConsumedLoopMux*. Only defined for edges leading to nodes of type loopMux.

***SetCancelConsumed*(*y*, *z*) :⇔**

| | |
|---|---|
| (1) If $z$ is neither a mux nor a loopMux, *SetCancelConsumedStandard* is fulfilled. | $t(z) \notin \{mux, loopMux\} \Rightarrow$<br>$SetCancelConsumedStandard(y, z)$ |
| $\wedge$ | |
| (2) If $z$ is a mux, *SetCancelConsumedMux* is fulfilled. | $t(z) = mux \Rightarrow$<br>$SetCancelConsumedMux(y, z)$ |
| $\wedge$ | |
| (3) If $z$ is a loopMux, *SetCancelConsumedLoopMux* is fulfilled. | $t(z) = loopMux \Rightarrow$<br>$SetCancelConsumedLoopMux(y, z)$ |

Table A.48.: Definition of condition *SetCancelConsumed*.

***ResetCancelConsumed*$(y, z)$ :⇔**

| | |
|---|---|
| (1) If $(y, z)$ is a data edge,<br><br>$z$'s *CancelState* is reset. | $(y, z) \in E_{dat} \Rightarrow$<br><br>$[(t(z) \notin \{mux, loopMux\} \Rightarrow$<br>$CancelStateAck(z)) \wedge$<br>$(t(z) \in \{mux, loopMux\} \Rightarrow$<br>$CancelStateAck(y, z))]$ |
| $\wedge$ | |
| (2) If $(y, z)$ is a control edge,<br><br>$y$'s *CancelStateCtrl* is acknowledged. | $(y, z) \in E_{con} \Rightarrow$<br><br>$CancelStateCtrlAck(y)$ |

Table A.49.: Definition of condition *ResetCancelConsumed*.

# B. CMDFG Node Types for C Operations and Statements

Table B.1 shows a complete list of assignments from C operations and statements to CMDFG node types created during the CMDFG generation pass (cf. Chapter 8).

The node created from a phi statement is an any node if the phi is used to merge memory flow using the virtual memory variable MEM_ACCESS. Otherwise, a loopMux is created if the phi statement is located in a loop header CFG node. If the originating CFG node is not a loop header, a mux is created instead.

Apart from the nodes listed in Table B.1, COMRADE creates an always node as token source for const nodes.

If there are multiple HW/SW transitions, an irqDataMux is created, which forwards the IRQ code of the particular transition to the irqreg.

Further nodes (and, any, nop) are inserted during the control edge creation phase.

| C statement or expression | CMDFG node type |
|---|---|
| a + b | add |
| a - b | sub |
| a * b | mul |
| a / b | div |
| a % b | mod |
| a << b | shiftLeft |
| a >> b | shiftRight |
| c = (unsigned char)a | bitSel (applies to other integer casts accordingly) |
| a < b | lessThan |
| a <= b | lessThanOrEqual |
| a > b | greaterThan |
| a >= b | greaterThanOrEqual |
| a == b | equal |
| a ! = b | notEqual |
| a & b | bitwiseAnd |
| a && b | logicalAnd |
| a \| b | bitwiseOr |
| a \|\| b | logicalOr |
| a^b | bitwiseXor |
| a = !b | negation |
| a = ~b | complement |
| a = *p | load |
| *p = a | store (if $p \neq 0$) |
| *0 = 0 | mf (memory forwarder) |
| 0 | const (analogous for other constants) |
| c = phi(a, b) | mux, loopMux, or any |
| LoadUp | inreg |
| PushDown | outreg |
| SignalIrq | irqreg |
| InitialMem | initial |

Table B.1.: Left: originating C operation or statement; right: node type of the created CMDFG node.

# C. Regression Tests

## C.1. Source Codes

```
1  int main() {
2      int i, s;
3
4      s = 0;
5
6      for (i = 0; i < 10; ++i) {
7          s += i;
8      }
9
10     printf("s = %d\n", s);
11     return 0;
12 }
```

Listing C.1: hg_regression_01

```
1  int main() {
2      int i, a[10];
3
4      a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
5      a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;
6
7      for (i = 0; i < 10; ++i) {
8          a[i] += 1;
9      }
10
11     printf("a[9] = %d\n", a[9]);
12     return 0;
13 }
```

Listing C.2: hg_regression_02

```
1  int main() {
2      int i, s;
3
4      s = 0;
5
6      for (i = 0; i < 10; ++i) {
7          if (i == 5) {
8              s += i;
9          }
10     }
11
12     printf("s = %d\n", s);
13     return 0;
14 }
```

Listing C.3: hg_regression_03

```
1  int main() {
2      int i, a[10];
3
4      a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
5      a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;
6
7      for (i = 0; i < 10; ++i) {
8          a[i] += 1;
9          if (i == 5) {
10             i += 1;
11         }
12     }
13
14     printf("a[9] = %d\n", a[9]);
15     return 0;
16 }
```

Listing C.4: hg_regression_04

```
1  int main() {
2      int i, a[10];
3
4      a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
5      a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;
```

```
6
7       for (i = 0; i < 10; ++i) {
8           if (i == 5) {
9               a[i] += 1;
10          }
11      }
12
13      printf("a[9] = %d\n", a[9]);
14      return 0;
15  }
```

Listing C.5: hg_regression_05

```
1   int main() {
2       int i, a[10];
3
4       a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
5       a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;
6
7       for (i = 0; i < 10; ++i) {
8           if (i == 5) {
9               i += 1;
10          }
11          a[i] += 1;
12      }
13
14      printf("a[9] = %d\n", a[9]);
15      return 0;
16  }
```

Listing C.6: hg_regression_06

```
1   int main() {
2       int i, a[10];
3
4       a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
5       a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;
6
7       for (i = 0; i < 10; ++i) {
8           a[i] += 1;
9           if (i == 5) {
```

```
10              a[i] += 2;
11          } else {
12              a[i] += 1;
13          }
14          a[i] += 1;
15      }
16
17      printf("a[9] = %d\n", a[9]);
18      return 0;
19  }
```

Listing C.7: hg_regression_07

```
1  int main() {
2      int i, s;
3
4      s = 0;
5
6      for (i = 0; i < 10; ++i) {
7          if (i > 5) {
8              if (i == 7) {
9                  s += 2;
10             } else {
11                 s += 1;
12             }
13         } else {
14             s += i;
15         }
16     }
17
18     printf("s = %d\n", s);
19     return 0;
20 }
```

Listing C.8: hg_regression_08

```
1  int main() {
2      int i, a[11];
3
4      a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
5      a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;
```

```
6       a[10] = 0;
7
8       for (i = 0; i < 10; ++i) {
9           a[i] += 1;
10          if (i > 5) {
11              a[i] += 2;
12              if (i == 7) {
13                  a[i] += 2;
14              } else {
15                  a[i] += 1;
16              }
17              a[i] += 1;
18          } else {
19              i += 1;
20          }
21          a[i] += 1;
22      }
23
24      printf("a[9] = %d\n", a[9]);
25      return 0;
26  }
```

Listing C.9: hg_regression_09

```
1   int main() {
2       int i, j, s;
3
4       s = 0;
5
6       for (i = 0; i < 10; ++i) {
7           for (j = 0; j < 3; ++j) {
8               s += j;
9           }
10      }
11
12      printf("s = %d\n", s);
13      return 0;
14  }
```

Listing C.10: hg_regression_10

```
1   int main() {
```

```
2       int i, j, a[10];
3
4       a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
5       a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;
6
7       for (i = 0; i < 10; ++i) {
8           a[i] += 1;
9           for (j = 0; j < 3; ++j) {
10              a[i] += j;
11          }
12          a[i] += 1;
13      }
14
15      printf("a[9] = %d\n", a[9]);
16      return 0;
17  }
```

Listing C.11: hg_regression_11

```
1   int main() {
2       int i, j, s;
3
4       s = 0;
5
6       for (i = 0; i < 10; ++i) {
7           if (i == 5) {
8               for (j = 0; j < 3; ++j) {
9                   s += j;
10              }
11          }
12      }
13
14      printf("s = %d\n", s);
15      return 0;
16  }
```

Listing C.12: hg_regression_12

```
1   int main() {
2       int i, j, a[10];
3
```

```
    a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
    a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;

    for (i = 0; i < 10; ++i) {
        a[i] += 1;
        if (i > 5) {
            a[i] += 2;
            for (j = 0; j < 3; ++j) {
                a[i] += j;
            }
            a[i] += 3;
        }
        a[i] += 4;
    }

    printf("a[9] = %d\n", a[9]);
    return 0;
}
```

Listing C.13: hg_regression_13

```
int main() {
    int i, j, k, s, t;

    s = 0;
    t = 0;

    for (i = 0; i < 10; ++i) {
        for (j = 0; j < 3; ++j) {
            s += j;
        }
        for (k = 0; k < 3; ++k) {
            t += 2 * k;
        }
    }

    printf("s = %d\n", s);
    printf("t = %d\n", t);
    return 0;
```

```
}
```

Listing C.14: hg_regression_14

```
int main() {
    int i, j, k, a[10];

    a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
    a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;

    for (i = 0; i < 10; ++i) {
        a[i] += 1;
        for (j = 0; j < 3; ++j) {
            a[i] += j;
        }
        a[i] += 2;
        for (k = 0; k < 3; ++k) {
            a[i] += k;
        }
        a[i] += 3;
    }

    printf("a[9] = %d\n", a[9]);
    return 0;
}
```

Listing C.15: hg_regression_15

```
int main() {
    int i, j, k, s, t;

    s = 0;
    t = 0;

    for (i = 0; i < 10; ++i) {
        if (i > 5) {
            for (j = 0; j < 3; ++j) {
                s += j;
            }
        }
        for (k = 0; k < 3; ++k) {
```

```
14                 t += 2 * k;
15             }
16         }
17
18         printf("s = %d\n", s);
19         printf("t = %d\n", t);
20         return 0;
21 }
```

Listing C.16: hg_regression_16

```
1  int main() {
2      int i, j, k, a[10];
3
4      a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0;
5      a[5] = 0; a[6] = 0; a[7] = 0; a[8] = 0; a[9] = 0;
6
7      for (i = 0; i < 10; ++i) {
8          a[i] += 1;
9          if (i > 5) {
10             a[i] += 2;
11             for (j = 0; j < 3; ++j) {
12                 a[i] += j;
13             }
14             a[i] += 3;
15         }
16         a[i] += 4;
17         for (k = 0; k < 3; ++k) {
18             a[i] += k;
19         }
20         a[i] += 5;
21     }
22
23     printf("a[9] = %d\n", a[9]);
24     return 0;
25 }
```

Listing C.17: hg_regression_17

| | Dynamic CTs | | Static CTs | |
|---|---|---|---|---|
| | QDEPTH=0 | QDEPTH=16 | QDEPTH=0 | QDEPTH=16 |
| hg_regression_01 | 47 | 47 | 47 | 47 |
| hg_regression_02 | 128 | 128 | 128 | 128 |
| hg_regression_03 | 57 | 57 | 57 | 57 |
| hg_regression_04 | 138 | 138 | 138 | 138 |
| hg_regression_05 | 121 | 112 | 121 | 112 |
| hg_regression_06 | 158 | 158 | 158 | 158 |
| hg_regression_07 | 238 | 238 | 238 | 238 |
| hg_regression_08 | 65 | 65 | 65 | 65 |
| hg_regression_09 | 225 | 225 | 225 | 225 |
| hg_regression_10 | 185 | 185 | 176 | 176 |
| hg_regression_11 | 418 | 388 | 418 | 388 |
| hg_regression_12 | 70 | 69 | 70 | 69 |
| hg_regression_13 | 342 | 330 | 342 | 330 |
| hg_regression_14 | 185 | 185 | 176 | 176 |
| hg_regression_15 | 708 | 648 | 708 | 648 |
| hg_regression_16 | 185 | 185 | 180 | 177 |
| hg_regression_17 | 632 | 590 | 632 | 590 |

Table C.1.: Simulation runtimes of regression test hardware kernels in #cycles.

## C.2. Simulation Results

Table C.1 shows the measured number of simulation cycles for the kernels given in Section C.1, both for dynamic and static CTs, as well as for transparent output queues (16 entries, i.e., QDEPTH=16) and without output queues (QDEPTH=0). These measurements give the pure runtime of the hardware kernel including cache stalls on the adaptive computer ACE-M5, but excluding the time for the FPGA configuration and the transfer of variables between software and hardware.

# D. C Sources

## D.1. Fcdf22

Fcdf22, taken from the Versatility Honeywell Stressmark [Hone97], implements the kernel of a wavelet image transformation. We have unrolled the loop by factor two to increase the size of the loop body. In our measurements, we transform one pixel row of 256 pixels.

```
1  #define ROW 256
2
3  int main() {
4      int mid = ROW / 2 - 1;
5      int i, i1;
6      int s[ROW / 2], d[ROW / 2];
7      int s_i, s_i1, d_i, d_i1;
8
9      // initialize arrays s[], d[]
10
11     d[0]=d[0]+d[0]-s[0]-s[1];
12     s[0]=s[0]+((d[0]+d[0])>>3);
13
14     // HW starts
15     for (i = 1; i < mid; i += 2) {
16         i1 = i + 1;
17         d_i=d[i]+d[i]-s[i]-s[i1];
18         s_i=s[i]+((d[i-1]+d_i)>>3);
19
20         d_i1=d[i1]+d[i1]-s[i1]-s[i+2];
21         s_i1=s[i1]+((d_i+d_i1)>>3);
22
23         s[i] = s_i;
24         d[i] = d_i;
25         s[i1]= s_i1;
```

```
26          d[i1] = d_i1;
27      }
28      // HW ends
29
30      // output results s[], d[]
31
32      return 0;
33  }
```

Listing D.1: fcdf22

## D.2. GfMultiply

GfMultiply is a kernel of the pegwit program from the MediaBench suite [LePM97], performing elliptic curve cryptography.

```
1  // surrounding code omitted
2
3  // HW starts
4  for (j = q[0]; j; j--) {
5      if ((log_qj = lg[j]) != TOGGLE) {
6          x = log_pi + log_qj;
7          if (x >= TOGGLE) {
8              r[i+j-1] ^= expt[x - TOGGLE];
9          } else {
10             r[i+j-1] ^= expt[x];
11         }
12     }
13 }
14 // HW ends
```

Listing D.2: gfmultiply

## D.3. MD5

The MD5 kernel computes the MD5 hash value (128 bits) of a given message. The source code used has been initially implemented by Benjamin Thielmann and was then adjusted by ourselves for pipelining multiple messages.

```
1  // only HW kernel function printed here
2
3  void process_hw_loop(unsigned num_messages) {
4      // function block values
5      unsigned int a, b, c, d, temp, temp2, f, g;
6      // hashes from previous iteration
7      unsigned int h0_h, h1_h, h2_h, h3_h;
8      // counter variable
9      unsigned int h;
10
11     // _local pointers: tell compiler to use local
12     //                  BRAM
13     unsigned int* message_bram0_local
14             = message_bram0;
15     // ...
16     unsigned int* message_bram15_local
17             = message_bram15;
18     unsigned int* h0_local = h0;
19     unsigned int* h1_local = h1;
20     unsigned int* h2_local = h2;
21     unsigned int* h3_local = h3;
22
23     // "buffers" for message words
24     unsigned int t_var_0;
25     // ...
26     unsigned int t_var_15;
27
28     // HW starts
29     for(h=0; h<num_messages; h++) {
30         // init hash
31         a = h0_local[h];
32         b = h1_local[h];
33         c = h2_local[h];
34         d = h3_local[h];
35
36         // save previous hashes
37         h0_h = a;
38         h1_h = b;
39         h2_h = c;
40         h3_h = d;
```

```
41
42          // 16 message_bram_locals
43          // => together 16 words of one message
44          t_var_0 = message_bram0_local[h];
45          // ...
46          t_var_15 = message_bram15_local[h];
47
48          // function block 0
49          f = (d ^ (b & (c ^ d)));
50          temp = d ;
51          d  = c ;
52          c  = b ;
53          temp2 = a + f + (0xd76aa478 + t_var_0);
54          b  = b + ((temp2 << 7) | (temp2 >> (25)));
55          a  = temp;
56
57          // more function blocks...
58
59          // function block 63
60          f = c ^ (b | (0xffffffff - d));
61          temp = d ;
62          d  = c ;
63          c  = b ;
64          temp2 = a + f + (0xeb86d391 + t_var_9);
65          b  = b + ((temp2 << 21) | (temp2 >> 11));
66          a  = temp;
67
68          // add hash and store
69          h0_local[h] = h0_h + a;
70          h1_local[h] = h1_h + b;
71          h2_local[h] = h2_h + c;
72          h3_local[h] = h3_h + d;
73      }
74      // HW ends
75  }
```

Listing D.3: md5

## D.4. Memcopy

Memcopy copies data from one array to another. While this is a synthetic kernel, copying memory is a basic technique that occurs in many applications. To enlarge the loop body, we copy eight data words per loop iteration.

```
1  int main() {
2      int i, a0, a1, a2, a3, a4, a5, a6, a7;
3      int offset0, offset1, offset2, offset3,
4          offset4, offset5, offset6, offset7;
5      int a[64], b[64];
6      a[0] = 0; a[1] = 1; a[2] = 2;
7      /* ... */ a[63] = 64;
8      b[0] = 100; b[1] = 101; b[2] = 102;
9      /* ... */ b[63] = 164;
10     offset0 = 0; offset1 = 4; offset2 = 8;
11     /* ... */ offset7 = 28;
12
13     // HW starts
14     for (i = 0; i < 64; i += 8) {
15         a0 = *((int*)(((int)a) + offset0));
16         // ...
17         a7 = *((int*)(((int)a) + offset7));
18
19         *((int*)(((int)b) + offset0)) = a0;
20         // ...
21         *((int*)(((int)b) + offset7)) = a7;
22
23         offset0 += 32;
24         // ...
25         offset7 += 32;
26     }
27     // HW ends
28
29     printf("result: b[0] = %d\n", b[0]);
30     // ...
31     printf("result: b[63] = %d\n", b[63]);
32     return 0;
33 }
```

Listing D.4: memcopy

## D.5. Quantization

The quantization kernel, taken from the Versatility Honeywell Stressmark [Hone97], quantizes the output of a wavelet image transform (32 bits per pixel) to four bits per pixel. We have replaced the call to the C function `abs` (the source code of which not being known to the compiler) by an if/else construct. For better efficiency, we iterate over pixel rows instead of columns in the outer loop. For our measurements, we use an input image area of 64x64 pixels.

```
1  // surrounding code omitted
2
3  // HW starts
4  for(j=y; j<y+size; j++) {
5      for(i=x; i<x+size; i++) {
6
7  val=my_int_data[i+(j<<9)];
8  if (val >= 0) {
9      absval = val;
10 } else {
11     absval = 0 - val;
12 }
13 if (absval < my_blockthresh[num]) {
14     result_val=ZERO_MARK;
15 } else {
16     // classify
17     if(val>mythresh8) {
18         if(val>mythresh12) {
19             if(val>mythresh14) {
20                 if(val>mythresh15) classify_val =15;
21                 else classify_val = 14;
22             } else {
23                 if(val>mythresh13) classify_val =13;
24                 else classify_val = 12;
25             }
26         } else {
27             if(val>mythresh10) {
28                 if(val>mythresh11) classify_val =11;
29                 else classify_val = 10;
30             } else {
31                 if(val>mythresh9 ) classify_val = 9;
```

```
32                  else classify_val = 8;
33              }
34          }
35
36      } else {
37          if(val>mythresh4) {
38              if(val>mythresh6 ) {
39                  if(val>mythresh7 ) classify_val = 7;
40                  else classify_val = 6;
41              } else {
42                  if(val>mythresh5 ) classify_val = 5;
43                  else classify_val = 4;
44              }
45          } else {
46              if(val>mythresh2 ) {
47                  if(val>mythresh3 ) classify_val = 3;
48                  else classify_val = 2;
49              } else {
50                  if(val>mythresh1 ) classify_val = 1;
51                  else classify_val = 0;
52              }
53          }
54      }
55      result_val=classify_val;
56  }
57  my_quant_buf[num*IMG_SIZE+qsize]=result_val;
58  ++qsize;
59
60      }
61  }
62  // HW ends
```

Listing D.5: quantization

## D.6. SHA

SHA from CHStone [HTHT09] represents a part of the computation of the SHA hash value of a given message. We have unrolled the loop by factor three to enlarge the loop body.

```
// surrounding code omitted

// HW starts
for (i = 16; i < 79; i = i + 3) {
    iM14 = W[i - 14];
    tmp1 = W[i - 3] ^ W[i - 8] ^ iM14 ^ W[i - 16];
    tmp2 = W[i - 2] ^ W[i - 7] ^
           W[i - 13] ^ W[i - 15];
    tmp3 = W[i - 1] ^ W[i - 6] ^ W[i - 12] ^ iM14;

    W[i] = tmp1;
    W[i+1] = tmp2;
    W[i+2] = tmp3;
}
// HW ends
W[79] = W[79 - 3] ^ W[79 - 8] ^
        W[79 - 14] ^ W[79 - 16];
```

Listing D.6: sha

## D.7. Susan

Susan is taken from MiBench [GREA01] and performs an edge detection on a gray scale image. For our measurements, we use an input image block of 8x8 pixels. In the inner loop, we have replaced the long addition chain – caused by the original p++ expressions – by a more compact address computation. To reduce the time for computing n, we have replaced the 36 serial additions by a tree adder network (using helper variables a1, ..., a36).

```
// surrounding code omitted

susan_principle(in,r,bp,max_no,x_size,y_size)
    uchar *in, *bp;
    int   *r, max_no, x_size, y_size;
{
   int   i, j, n;
   uchar *p,*cp;
   int a1, a2, a3, a4, a5, a6, a7, a8,
       a9, a10, a11, a12, a13, a14, a15, a16,
```

```
11          a17, a18, a19, a20, a21, a22, a23, a24,
12          a25, a26, a27, a28, a29, a30, a31, a32,
13          a33, a34, a35, a36;
14
15      memset (r,0,x_size * y_size * sizeof(int));
16
17      // HW starts
18      for (i=3;i<y_size-3;i++) {
19          for (j=3;j<x_size-3;j++) {
20              n=100;
21              p=in + (i-3)*x_size + j - 1;
22              cp=bp + in[i*x_size+j];
23
24              // replaced "p++" for more compact
25              // address computations
26              a1=*(cp-*p);
27              a2=*(cp-*(p+1));
28              a3=*(cp-*(p+2));
29              a4=*(cp-*(p+x_size-1));
30              p+=x_size;
31
32              a5=*(cp-*p);
33              a6=*(cp-*(p+1));
34              a7=*(cp-*(p+2));
35              a8=*(cp-*(p+3));
36              p+=x_size-2;
37
38              a9=*(cp-*p);
39              a10=*(cp-*(p+1));
40              a11=*(cp-*(p+2));
41              a12=*(cp-*(p+3));
42              p+=4;
43
44              a13=*(cp-*p);
45              a14=*(cp-*(p+1));
46              a15=*(cp-*(p+2));
47              a16=*(cp-*(p+x_size-4));
48              p+=x_size-3;
49
50              a17=*(cp-*p);
```

```
51          a18=*(cp-*(p+1));
52          a19=*(cp-*(p+3));
53          a20=*(cp-*(p+4));
54          p+=5;
55
56          a21=*(cp-*p);
57          a22=*(cp-*(p+x_size-6));
58          a23=*(cp-*(p+x_size-5));
59          a24=*(cp-*(p+x_size-4));
60          p+=x_size-3;
61
62          a25=*(cp-*p);
63          a26=*(cp-*(p+1));
64          a27=*(cp-*(p+2));
65          a28=*(cp-*(p+3));
66          p+=x_size-2;
67
68          a29=*(cp-*p);
69          a30=*(cp-*(p+1));
70          a31=*(cp-*(p+2));
71          a32=*(cp-*(p+3));
72          p+=4;
73
74          a33=*(cp-*p);
75          a34=*(cp-*(p+x_size-3));
76          a35=*(cp-*(p+x_size-2));
77          a36=*(cp-*(p+x_size-1));
78
79          // tree adder for n
80          a1 += a2;    a3 += a4;
81          a5 += a6;    a7 += a8;
82          a9 += a10;   a11 += a12;
83          a13 += a14;  a15 += a16;
84          a17 += a18;  a19 += a20;
85          a21 += a22;  a23 += a24;
86          a25 += a26;  a27 += a28;
87          a29 += a30;  a31 += a32;
88          a33 += a34;  a35 += a36;
89
90          a1 += a3;    a5 += a7;
```

```
91              a9 += a11;   a13 += a15;
92              a17 += a19;  a21 += a23;
93              a25 += a27;  a29 += a31;
94              a33 += a35;
95
96              a1 += a5;    a9 += a13;
97              a17 += a21;  a25 += a29 + a33;
98
99              a1 += a9;
100             a17 += a25;
101
102             a1 += a17;
103
104             n += a1;
105
106             if (n<=max_no) {
107                 r[i*x_size+j] = max_no - n;
108             }
109         }
110     }
111     // HW ends
112 }
```

Listing D.7: susan

## D.8. Fcdf22_local

Fcdf22_local is an adjusted version of the wavelet transform function fcdf22 offered by the Versatility Honeywell Stressmark [Hone97]. We use it for the application-level study in Chapter 10.

```
1  void fcdf22_local(int line_size, int num_blocks) {
2      int i, h1, h2, h3, h4, h5, h6, f1, f2;
3      int pixel;
4
5      f1 = 0;
6      pixel = 0;
7
8      // HW kernel start
```

```
9       for (i = 0; i < num_blocks; i++) {
10          // iterate over 16 pixels
11
12          f2 = s_pixel_0_bram0_local[i];
13
14          // === pixels 0, 1 ===
15          h1 = d_pixel_0_bram1_local[i]; h2 = f2;
16          h3 = s_pixel_1_bram2_local[i];
17          h4 = f1;
18
19          h5 = (h1 << 1) - (h2 + h3);
20
21          if (pixel == 0) {
22              // line start
23              h6 = h2 + ((h5 << 1) >> 3);
24          } else {
25              // line middle
26              h6 = h2 + ((h4 + h5) >> 3);
27          }
28
29          d_pixel_0_bram1_local[i] = h5;
30          s_pixel_0_bram0_local[i] = h6;
31
32          f1 = h5;
33          f2 = h3;
34
35          // === pixels 2, 3 ===
36          h1 = d_pixel_1_bram3_local[i]; h2 = f2;
37          h3 = s_pixel_2_bram4_local[i];
38          h4 = f1;
39
40          h5 = (h1 << 1) - (h2 + h3);
41          h6 = h2 + ((h4 + h5) >> 3);
42
43          d_pixel_1_bram3_local[i] = h5;
44          s_pixel_1_bram2_local[i] = h6;
45
46          f1 = h5;
47          f2 = h3;
48
```

```
49          // === pixels 4, 5 ===
50          h1 = d_pixel_2_bram5_local[i]; h2 = f2;
51          h3 = s_pixel_3_bram6_local[i];
52          h4 = f1;
53
54          h5 = (h1 << 1) - (h2 + h3);
55          h6 = h2 + ((h4 + h5) >> 3);
56
57          d_pixel_2_bram5_local[i] = h5;
58          s_pixel_2_bram4_local[i] = h6;
59
60          f1 = h5;
61          f2 = h3;
62
63          // === pixels 6, 7 ===
64          h1 = d_pixel_3_bram7_local[i]; h2 = f2;
65          h3 = s_pixel_4_bram8_local[i];
66          h4 = f1;
67
68          h5 = (h1 << 1) - (h2 + h3);
69          h6 = h2 + ((h4 + h5) >> 3);
70
71          d_pixel_3_bram7_local[i] = h5;
72          s_pixel_3_bram6_local[i] = h6;
73
74          f1 = h5;
75          f2 = h3;
76
77          // === pixels 8, 9 ===
78          h1 = d_pixel_4_bram9_local[i]; h2 = f2;
79          h3 = s_pixel_5_bram10_local[i];
80          h4 = f1;
81
82          h5 = (h1 << 1) - (h2 + h3);
83          h6 = h2 + ((h4 + h5) >> 3);
84
85          d_pixel_4_bram9_local[i] = h5;
86          s_pixel_4_bram8_local[i] = h6;
87
88          f1 = h5;
```

```
89          f2 = h3;
90
91          // === pixels 10, 11 ===
92          h1 = d_pixel_5_bram11_local[i]; h2 = f2;
93          h3 = s_pixel_6_bram12_local[i];
94          h4 = f1;
95
96          h5 = (h1 << 1) - (h2 + h3);
97          h6 = h2 + ((h4 + h5) >> 3);
98
99          d_pixel_5_bram11_local[i] = h5;
100         s_pixel_5_bram10_local[i] = h6;
101
102         f1 = h5;
103         f2 = h3;
104
105         // === pixels 12, 13 ===
106         h1 = d_pixel_6_bram13_local[i]; h2 = f2;
107         h3 = s_pixel_7_bram14_local[i];
108         h4 = f1;
109
110         h5 = (h1 << 1) - (h2 + h3);
111         h6 = h2 + ((h4 + h5) >> 3);
112
113         d_pixel_6_bram13_local[i] = h5;
114         s_pixel_6_bram12_local[i] = h6;
115
116         f1 = h5;
117         f2 = h3;
118
119         // === pixels 14, 15 ===
120         h1 = d_pixel_7_bram15_local[i]; h2 = f2;
121         h3 = s_pixel_8_bram16_local[i];
122         h4 = f1;
123
124         if (pixel < line_size - 16) {
125             // line middle
126             h5 = (h1 << 1) - (h2 + h3);
127         } else {
128             // line end
```

```
129                 h5 = (h1 << 1) - (h2 << 1);
130             }
131
132             h6 = h2 + ((h4 + h5) >> 3);
133
134             d_pixel_7_bram15_local[i] = h5;
135             s_pixel_7_bram14_local[i] = h6;
136             f1 = h5;
137
138             pixel += 16;
139             if (pixel >= line_size) {
140                 pixel = 0;
141             }
142         }
143         // HW kernel ends
144 }
```

Listing D.8: fcdf22_local

# Curriculum Vitae

## Personal Data

Name: Hagen Gädke-Lütjens

Born: 15 Febuary 1979 in Berlin, Germany

Marital Status: married

## Education

| | |
|---|---|
| 1991 – 1998 | Gymnasium Alfeld in Alfeld (Leine) |
| July 1998 | Abitur at Gymnasium Alfeld |
| 1999 – 2004 | Study of Computer Science at Technische Universität Braunschweig, Germany |
| October 2004 | Diplom in Computer Science |

## Compulsory Community Service

| | |
|---|---|
| 1998 – 1999 | DRK Tagespflege-Einrichtung in Gronau (Leine) |

## Work Experience: Academic

| | |
|---|---|
| 2000 – 2001 | Student developer at the Fraunhofer-Institute for Surface Engineering and Thin Films in Braunschweig |
| 2001 – 2003 | Student developer at the Computing Center, Technische Universität Braunschweig |
| 2003 – 2004 | Student developer at the Department for Computer Graphics, Technische Universität Braunschweig |
| 2005 – 2011 | Research Assistant at the Department for VLSI Design, Technische Universität Braunschweig |

## Work Experience: Commercial

| | |
|---|---|
| 2004 – 2005 | Internship at Siemens Signalling Company Ltd. in Xi'an, P. R. China |

# Lebenslauf

## Persönliche Daten

Name: Hagen Gädke-Lütjens

Geboren am 15. Februar 1979 in Berlin

Familienstand: verheiratet

## Ausbildung

| | |
|---|---|
| 1991 – 1998 | Gymnasium Alfeld in Alfeld (Leine) |
| Juli 1998 | Abitur am Gymnasium Alfeld |
| 1999 – 2004 | Studium der Informatik an der Technischen Universität Braunschweig |
| Oktober 2004 | Diplom in Informatik |

## Zivildienst

| | |
|---|---|
| 1998 – 1999 | DRK Tagespflege-Einrichtung in Gronau (Leine) |

## Berufstätigkeit im akademischen Bereich

| | |
|---|---|
| 2000 – 2001 | Studentische Hilfskraft am Fraunhofer-Institut für Schicht- und Oberflächentechnik (IST) in Braunschweig |
| 2001 – 2003 | Studentische Hilfskraft am Rechenzentrum der Technischen Universität Braunschweig |
| 2003 – 2004 | Studentische Hilfskraft am Institut für Computergraphik, Technische Universität Braunschweig |
| 2005 – 2011 | Wissenschaftlicher Mitarbeiter an der Abteilung Entwurf integrierter Schaltungen (E.I.S.), Technische Universität Braunschweig |

## Berufstätigkeit im kommerziellen Bereich

| | |
|---|---|
| 2004 – 2005 | Praktikum in der Siemens Signalling Company Ltd. in Xi'an, V.R. China |